

conference

proceedings

# General Track

## 2002 USENIX Annual Technical Conference

*Monterey, California, USA*  
*June 10–15, 2002*

Sponsored by  
The USENIX Association

**USENIX**  
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

General Track: 2002 USENIX Annual Technical Conference

Monterey, California, USA, June 2002

For additional copies of these proceedings contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA  
Phone: 510 528 8649  
FAX: 510 548 5738  
Email: [office@usenix.org](mailto:office@usenix.org)  
WWW URL: <http://www.usenix.org>

The price is \$35 for members and \$45 for nonmembers.  
Outside the U.S.A. and Canada, please add  
\$18 per copy for postage (via air printed matter).

### **Past USENIX Technical Conferences**

2001 Boston, MA	1990 Winter Washington, D.C.
2000 San Diego, CA	1989 Summer Baltimore
1999 Monterey CA	1989 Winter San Diego
1998 New Orleans	1988 Summer San Francisco
1997 Anaheim	1988 Winter Dallas
1996 San Diego	1987 Summer Phoenix
1995 New Orleans	1987 Winter Washington, D.C.
1994 Summer Boston	1986 Summer Atlanta
1994 Winter San Francisco	1986 Winter Denver
1993 Summer Cincinnati	1985 Summer Portland
1993 Winter San Diego	1985 Winter Dallas
1992 Summer San Antonio	1984 Summer Salt Lake City
1992 Winter San Francisco	1984 Winter Washington, D.C.
1991 Summer Nashville	1983 Summer Toronto
1991 Winter Dallas	1983 Winter San Diego
1990 Summer Anaheim	

© 2002 by The USENIX Association  
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-880446-00-6

Printed in the United States of America on 50% recycled paper, 10–15% post consumer waste.



**USENIX Association**

**Proceedings of the  
General Track**

**2002 USENIX Annual Technical Conference**

**June 10–15, 2002  
Monterey, California, USA**

## Conference Organizers

### Program Chair

Carla Ellis, *Duke University*

### Program Committee

Darrell Anderson, *Duke University*

Mary Baker, *Stanford University*

Frank Bellosa, *University of Erlangen-Nürnberg*

Greg Ganger, *Carnegie Mellon University*

Mike Jones, *Microsoft Research*

Patrick McDaniel, *AT&T Labs—Research*

Jason Nieh, *Columbia University*

Vern Paxson, *ACIRI*

Elizabeth Shriver, *Bell Labs*

Christopher Small, *Sun Microsystems*

Mirjana Spasojevic, *Hewlett Packard*

Mike Spreitzer, *IBM*

Amin Vahdat, *Duke University*

### Invited Talks Coordinators

Matt Blaze, *AT&T Labs—Research*

Ted Faber, *USC Information Sciences Institute*

### “The Guru is In” Coordinator

Lee Damon, *University of Washington*

### The USENIX Association Staff

## External Reviewers

Lisa Amini

Martin Arlitt

John Barton

Steve Bellovin

Nina Bhatti

Rebecca Braynard

Lucy Cherkasova

Norman H. Cohen

Alan Cole

Brendan Connell

Alexandra Fedorova

Rick Floyd

Sally Floyd

Liana L. Fong

Yun Fu

Eran Gabber

Richard Golding

Pankaj Gupta

Ben Jai

Kevin Jeffay

Wenyu Jiang

William Josephson

Michael Kalantar

Scott F. Kaplan

Kimberly Keeton

Geoffrey H. Kuenning

Jonathan Ledlie

Hsin-Ting (Jerry) Liu

James Lujan

Georgi Matev

Dejan Milojicic

Pia Mukherji

Jonathan Munson

Nicholas C. Murphy

Wee Teck Ng

David Olshefski

Giovanni Pacifici

Maria Papadopoulou

Craig Partridge

Arnold G. Paul

Marco Pistoia

Patrick Reynolds

Erik Riedel

Wolfgang Schroeder-Preikschat

Wolfgang Segmuller

Keith A. Smith

Olaf Spinczyk

Guy L. Steele

Gong Su

Ram Swaminathan

Asser N. Tantawi

Dafina Toncheva

Mark Tonkelowitz

George Varghese

Michael Vernal

Dario Vlah

Ken Yocum

Alaa Youssef

Kan Zhang

Haoqiang Zheng

**2002 USENIX Annual Technical Conference**  
**General Track**  
**June 10–15, 2002**  
**Monterey, California, USA**

<b>Index of Authors</b> .....	vii
<b>Message from the Program Chair</b> .....	ix

**Thursday, June 13**

**File Systems**

*Session Chair: Greg Ganger, Carnegie Mellon University*

Structure and Performance of the Direct Access File System .....	1
<i>Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, and Margo I. Seltzer, Harvard University; Jeffrey S. Chase, Andrew J. Gallatin, Richard Kisley, and Rajiv G. Wickremesinghe, Duke University; and Eran Gabber, Lucent Technologies</i>	

Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System .....	15
<i>An-I A. Wang, Peter Reiher, and Gerald J. Popek, University of California, Los Angeles; and Geoffrey H. Kuenning, Harvey Mudd College</i>	

Exploiting Gray-Box Knowledge of Buffer-Cache Management .....	29
<i>Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison</i>	

**Operating Systems (and Dancing Bears)**

*Session Chair: Frank Bellosa, University of Erlangen-Nürnberg*

The JX Operating System .....	45
<i>Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder, University of Erlangen-Nürnberg</i>	

Design Evolution of the EROS Single-Level Store .....	59
<i>Jonathan S. Shapiro, Johns Hopkins University; and Jonathan Adams, University of Pennsylvania</i>	

THINK: A Software Framework for Component-based Operating System Kernels .....	73
<i>Jean-Philippe Fassino, France Télécom R&amp;D; Jean-Bernard Stefani, INRIA; Julia Lawall, DIKU; and Gilles Muller, INRIA</i>	

**Building Services**

*Session Chair: Jason Nieh, Columbia University*

Ninja: A Framework for Network Services .....	87
<i>J. Robert von Behren, Eric A. Brewer, Nikita Borisov, Michael Chen, Matt Welsh, Josh MacDonald, Jeremy Lau, and David Culler, University of California, Berkeley</i>	

Using Cohort-Scheduling to Enhance Server Performance .....	103
<i>James R. Larus and Michael Parkes, Microsoft Research</i>	

## Friday, June 14

### Network Performance

*Session Chair: Vern Paxson, ACIRI*

EtE: Passive End-to-End Internet Service Performance Monitoring . . . . . 115  
*Yun Fu and Amin Vahdat, Duke University; Ludmila Cherkasova and Wenting Tang, Hewlett-Packard Laboratories*

The Performance of Remote Display Mechanisms for Thin-Client Computing . . . . . 131  
*S. Jae Yang, Jason Nieh, Matt Selsky, and Nikhil Tiwari, Columbia University*

A Mechanism for TCP-Friendly Transport-Level Protocol Coordination . . . . . 147  
*David E. Ott and Ketan Mayer-Patel, University of North Carolina, Chapel Hill*

### Storage Systems

*Session Chair: Elizabeth Shriver, Bell Labs*

My Cache or Yours? Making Storage More Exclusive . . . . . 161  
*Theodore M. Wong, Carnegie Mellon University; and John Wilkes, Hewlett-Packard Laboratories*

Bridging the Information Gap in Storage Protocol Stacks . . . . . 177  
*Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison*

Maximizing Throughput in Replicated Disk Striping of Variable Bit-Rate Streams . . . . . 191  
*Stergios V. Anastasiadis, Duke University; and Kenneth C. Sevcik and Michael Stumm, University of Toronto*

### Tools

*Session Chair: Christopher Small, Sun Microsystems*

Simple and General Statistical Profiling with PCT . . . . . 205  
*Charles Blake and Steven Bauer, Massachusetts Institute of Technology*

Engineering a Differencing and Compression Data Format . . . . . 219  
*David G. Korn and Kiem Phong Vo, AT&T Labs—Research*

## Saturday, June 15

### Where in the Net . . .

*Session Chair: Patrick McDaniel, AT&T Labs—Research*

A Precise and Efficient Evaluation of the Proximity Between Web Clients and Their Local DNS Servers . . . . . 229  
*Zhuoqing Morley Mao, University of California, Berkeley; and Charles D. Cranor, Fred Douglass, Michael Rabinovich, Oliver Spatscheck, and Jia Wang, AT&T Labs—Research*

Geographic Properties of Internet Routing . . . . . 243  
*Lakshminarayanan Subramanian, University of California, Berkeley; Venkata N. Padmanabhan, Microsoft Research; and Randy H. Katz, University of California, Berkeley*

Providing Process Origin Information to Aid in Network Traceback . . . . . 261  
*Florian P. Buchholz, Purdue University; and Clay Shields, Georgetown University*

## **Programming**

*Session Chair: Darrell Anderson, Duke University*

Cyclone: A Safe Dialect of C ..... 275  
*Trevor Jim, AT&T Labs—Research; and Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang, Cornell University*

Cooperative Task Management Without Manual Stack Management ..... 289  
*Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur, Microsoft Research*

Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems ..... 303  
*Hai Huang, Padmanabhan Pillai, and Kang G. Shin, University of Michigan*

## **Mobility**

*Session Chair: Mary Baker, Stanford University*

Robust Positioning Algorithms for Distributed Ad-Hoc Wireless Sensor Networks ..... 317  
*Chris Savarese and Jan Rabaey, University of California, Berkeley; Koen Langendoen, Delft University of Technology*

Application-specific Network Management for Energy-Aware Streaming of Popular Multimedia Formats ..... 329  
*Surender Chandra, University of Georgia; and Amin Vahdat, Duke University*

Characterizing Alert and Browse Services of Mobile Clients ..... 343  
*Atul Adya, Paramvir Bahl, and Lili Qiu, Microsoft Research*



## Index of Authors

Adams, Jonathan	59	Larus, James R.	103
Addetia, Salimah	1	Lau, Jeremy	87
Adya, Atul	289, 343	Lawall, Julia	73
Anastasiadis, Stergios V.	191	MacDonald, Josh	87
Arpaci-Dusseau, Andrea C.	29, 177	Magoutis, Kostas	1
Arpaci-Dusseau, Remzi H.	29, 177	Mao, Zhuoqing Morley	229
Bahl, Paramvir	343	Mayer-Patel, Ketan	147
Bauer, Steven	205	Morrisett, Greg	275
Bent, John	29	Muller, Gilles	73
Blake, Charles	205	Nieh, Jason	131
Bolosky, William J.	289	Ott, David E.	147
Borisov, Nikita	87	Padmanabhan, Venkata N.	243
Brewer, Eric A.	87	Parkes, Michael	103
Buchholz, Florian P.	261	Pillai, Padmanabhan	303
Burnett, Nathan C.	29	Popek, Gerald J.	15
Chandra, Surender	329	Qiu, Lili	343
Chase, Jeffrey S.	1	Rabaey, Jan	317
Chen, Michael	87	Rabinovich, Michael	229
Cheney, James	275	Reiher, Peter	15
Cherkasova, Ludmila	115	Savarese, Chris	317
Cranor, Charles D.	229	Selsky, Matt	131
Culler, David	87	Seltzer, Margo I.	1
Denehy, Timothy E.	177	Sevcik, Kenneth C.	191
Douceur, John R.	289	Shapiro, Jonathan S.	59
Douglis, Fred	229	Shields, Clay	261
Fassino, Jean-Philippe	73	Shin, Kang G.	303
Fedorova, Alexandra	1	Spatscheck, Oliver	229
Felser, Meik	45	Stefani, Jean-Bernard	73
Fu, Yun	115	Stumm, Michael	191
Gabber, Eran	1	Subramanian, Lakshminarayanan	243
Gallatin, Andrew J.	1	Tang, Wenting	115
Golm, Michael	45	Theimer, Marvin	289
Grossman, Dan	275	Tiwari, Nikhil	131
Hicks, Michael	275	Vahdat, Amin	115, 329
Howell, Jon	289	Vo, Kiem Phong	219
Huang, Hai	303	von Behren, J. Robert	87
Jim, Trevor	275	Wang, An-I A.	15
Katz, Randy H.	243	Wang, Jia	229
Kisley, Richard	1	Wang, Yanling	275
Kleinöder, Jürgen	45	Wawersich, Christian	45
Korn, David G.	219	Welsh, Matt	87
Kuenning, Geoffrey H.	15	Wickremesinghe, Rajiv G.	1
Langendoen, Koen	317	Wilkes, John	161





## Message from the Program Chair

As I write this message, it is rewarding for me to see the finishing touches going into the production of these proceedings. The 2002 USENIX Annual Technical Conference promises to be a technical success with the program we have to offer—thanks to the hard work that has gone into it by the authors, the reviewers, and the USENIX staff.

This year we experienced a significant increase in the number of submissions to the refereed technical track over previous years. This proceedings contains 25 papers that were identified as the best of the 105 full papers that were submitted. The members of the program committee worked very hard with the unexpected extra workload (and hardly ever complained). We produced over 475 written reviews with the help of 58 external reviewers. The program committee meeting was held in Durham, North Carolina, in January to select the program. It was a stimulating meeting spent debating the technical merits of the many interesting and wide-ranging topics. I especially appreciated the positive tone maintained in the meeting as the hours passed and we got down to the hardest decisions. In the end, we felt we could be proud of this program and excited about bringing it to you. Every accepted paper was assigned a shepherd to guide revisions and improve the quality of the final papers you see here.

One of the benefits of serving as Program Chair is the opportunity to work closely with so many wonderful people. Along with being experts in their own technical fields, the members of my program committee are volunteers who are dedicated, cooperative, constructive, and a joy to know. Program committee members served as referees, as shepherds, on the awards subcommittee, and as session chairs. I want to acknowledge their significant contributions to the strength of this conference—many thanks to each one of them.

I also want to thank all of the USENIX staff and board members for the help and advice they have given me. My appreciation goes to Mike Jones, who not only served as USENIX Board liaison but also participated fully on the program committee. I also want to acknowledge the FREENIX program chair, Chris Demetriou, and the Invited Talks Coordinators, Matt Blaze and Ted Faber, for their important roles in putting together tempting “competing” tracks. Finally, a very special thanks to Ellie Young and Jane-ellen Long for making sure things happened as they should, for their patience, and for making my life much easier during this past year of planning and preparation.

So enjoy this 2002 USENIX Annual Technical Conference in Monterey.

**Carla Ellis, Program Chair**



# Structure and Performance of the Direct Access File System

Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo I. Seltzer

*Division of Engineering and Applied Sciences, Harvard University*

Jeffrey S. Chase, Andrew J. Gallatin, Richard Kisley, Rajiv G. Wickremesinghe

*Department of Computer Science, Duke University*

Eran Gabber

*Lucent Technologies - Bell Laboratories*

## Abstract

The Direct Access File System (DAFS) is an emerging industrial standard for network-attached storage. DAFS takes advantage of new user-level network interface standards. This enables a *user-level file system* structure in which client-side functionality for remote data access resides in a library rather than in the kernel. This structure addresses longstanding performance problems stemming from weak integration of buffering layers in the network transport, kernel-based file systems and applications. The benefits of this architecture include lightweight, portable and asynchronous access to network storage and improved application control over data movement, caching and prefetching.

This paper explores the fundamental performance characteristics of a user-level file system structure based on DAFS. It presents experimental results from an open-source DAFS prototype and compares its performance to a kernel-based NFS implementation optimized for zero-copy data transfer. The results show that both systems can deliver file access throughput in excess of 100 MB/s, saturating network links with similar raw bandwidth. Lower client overhead in the DAFS configuration can improve application performance by up to 40% over optimized NFS when application processing and I/O demands are well-balanced.

## 1 Introduction

The performance of high-speed network storage systems is often limited by client overhead, such as memory copying, network access costs and protocol overhead [2, 8, 20, 29]. A related source of inefficiency stems from poor integration of applications and file system services; lack of control over kernel policies leads to problems such as

double caching, false prefetching and poor concurrency management [34]. As a result, databases and other performance-critical applications often bypass file systems in favor of raw block storage access. This sacrifices the benefits of the file system model, including ease of administration and safe sharing of resources and data. These problems have also motivated the design of radical operating system structures to allow application control over resource management [21, 31].

The recent emergence of commercial *direct-access transport* networks creates an opportunity to address these issues without changing operating systems in common use. These networks incorporate two defining features: *user-level networking* and *remote direct memory access* (RDMA). User-level networking allows safe network communication directly from user-mode applications, removing the kernel from the critical I/O path. RDMA allows the network adapter to reduce copy overhead by accessing application buffers directly.

The Direct Access File System (DAFS) [14] is a new standard for network-attached storage over direct-access transport networks. The DAFS protocol is based on the Network File System Version 4 protocol [32], with added protocol features for direct data transfer using RDMA, scatter/gather list I/O, reliable locking, command flow-control and session recovery. DAFS is designed to enable a *user-level file system client*: a DAFS client may run as an application library above the operating system kernel, with the kernel's role limited to basic network device support and memory management. This structure can improve performance, portability and reliability, and offer applications fully asynchronous I/O and more direct control over data movement and caching. Network Appliance and other network-attached storage vendors are planning DAFS interfaces for their products.

This paper explores the fundamental structural and performance characteristics of network file access using a user-level file system structure on a direct-access transport network with RDMA. We use DAFS as a basis for exploring these features since it is the first fully-specified file system protocol to support them. We describe DAFS-based client and server reference implementations for an open-source Unix system (FreeBSD) and report experimental results, comparing DAFS to a zero-copy NFS implementation. Our purpose is to illustrate the benefits and tradeoffs of these techniques to provide a basis for informed choices about deployment of DAFS-based systems and similar extensions to other network file protocols, such as NFS.

Our experiments explore the application properties that determine how RDMA and user-level file systems affect performance. For example, when a workload is balanced (i.e., the application simultaneously saturates the CPU and network link) DAFS delivers the most benefit compared to more traditional architectures. When workloads are limited by the disk, DAFS and more traditional network file systems behave comparably. Other workload factors such as metadata-intensity, I/O sizes, file sizes, and I/O access pattern also influence performance.

An important property of the user-level file system structure is that applications are no longer bound by the kernel's policies for file system buffering, caching and prefetching. The user-level file system structure and the DAFS API allow applications full control over file system access; however, the application can no longer benefit from shared kernel facilities for caching and prefetching. A secondary goal of our work is to show how *adaptation libraries* for specific classes of applications enable those applications to benefit from improved control and tighter integration with the file system, while reducing or eliminating the burden on application developers. We present experiments with two adaptation libraries for DAFS clients: Berkeley DB [28] and the TPIE external memory I/O toolkit [37]. These adaptation libraries provide the benefits of the user-level file system without requiring that applications be modified to use the DAFS API.

The layout of this paper is as follows. Section 2 summarizes the trends that motivated DAFS and user-level file systems and sets our study in context with previous work. Section 3 gives an overview of the salient features of the DAFS specifications, and Section 4 describes the DAFS reference implementation used in the experiments. Section 5 presents two example adaptation libraries, and Section 6 de-

scribes zero-copy, kernel-based NFS as an alternative to DAFS. Section 7 presents experimental results. We conclude in Section 8.

## 2 Background and Related Work

In this section, we discuss the previous work that lays the foundation for DAFS and provides the context for our experimental results. We begin with a discussion of the issues that limit performance in network storage systems and then discuss the two critical architectural features that we examine to attack performance bottlenecks: direct-access transports and user-level file systems.

### 2.1 Network Storage Performance

Network storage solutions can be categorized as Storage-Area Network (SAN)-based solutions, which provide a block abstraction to clients, and Network-Attached Storage (NAS)-based solutions, which export a network file system interface. Because a SAN storage volume appears as a local disk, the client has full control over the volume's data layout; client-side file systems or database software can run unmodified [23]. However, this precludes concurrent access to the shared volume from other clients, unless the client software is extended to coordinate its accesses with other clients [36]. In contrast, a NAS-based file service can control sharing and access for individual files on a shared volume. This approach allows safe data sharing across diverse clients and applications.

Communication overhead was a key factor driving acceptance of Fibre Channel [20] as a high-performance SAN. Fibre Channel leverages network interface controller (NIC) support to offload transport processing from the host and access I/O blocks in host memory directly without copying. Recently, NICs supporting the emerging iSCSI block storage standard have entered the market as an IP-based SAN alternative. In contrast, NAS solutions have typically used IP-based protocols over conventional NICs, and have paid a performance penalty. The most-often cited causes for poor performance of network file systems are (a) protocol processing in network stacks; (b) memory copies [2, 15, 29, 35]; and (c) other kernel overhead such as system calls and context switches. Data copying, in particular, incurs substantial per-byte overhead in the CPU and memory system that is not masked by advancing processor technology.

One way to reduce network storage access over-

head is to offload some or all of the transport protocol processing to the NIC. Many network adapters can compute Internet checksums as data moves to and from host memory; this approach is relatively simple and delivers a substantial benefit. An increasing number of adapters can offload all TCP or UDP protocol processing, but more substantial kernel revisions are needed to use them. Neither approach by itself avoids the fundamental overheads of data copying.

Several known techniques can remove copies from the transport data path. Previous work has explored copy avoidance for TCP/IP communication (Chase et al. [8] provide a summary). Brustoloni [5] introduced *emulated copy*, a scheme that avoids copying in network I/O while preserving copy semantics. IO-Lite [29] adds scatter/gather features to the I/O API and relies on support from the NIC to handle multiple client processes safely without copying. Another approach is to implement critical applications (e.g., Web servers) in the kernel [19]. Some of the advantages can be obtained more cleanly with combined data movement primitives, e.g., *sendfile*, which move data from storage directly to a network connection without a user space transfer; this is useful for file transfer in common server applications.

DAFS was introduced to combine the low overhead and flexibility of SAN products with the generality of NAS file services. The DAFS approach to removing these overheads is to use a direct-access transport to read and write application buffers directly. DAFS also enables implementation of the file system client at user level for improved efficiency, portability and application control. The next two sections discuss these aspects of DAFS in more detail. In Section 6, we discuss an alternative approach that reduces NFS overhead by eliminating data copying.

## 2.2 Direct-Access Transports

Direct-access transports are characterized by NIC support for remote direct memory access (RDMA), user-level networking with minimal kernel overhead, reliable messaging transport connections and per-connection buffering, and efficient asynchronous event notification. The Virtual Interface (VI) Architecture [12] defines a host interface and API for NICs supporting these features.

Direct-access transports enable *user-level networking* in which the user-mode process interacts directly with the NIC to send or receive messages

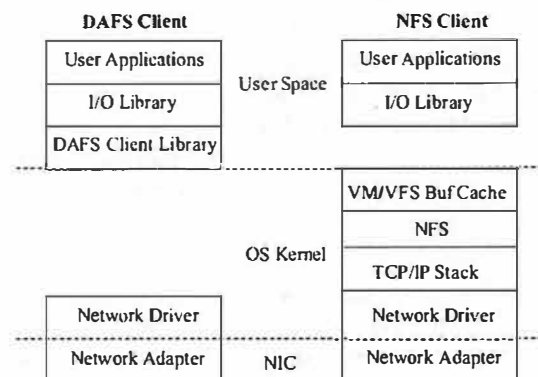


Figure 1: User-level vs. kernel-based client file system structure.

with minimal intervention from the operating system kernel. The NIC device exposes an array of connection descriptors to the system physical address space. At connection setup time, the kernel network driver maps a free connection descriptor into the user process virtual address space, giving the process direct and safe access to NIC control registers and buffer queues in the descriptor. This enables RDMA, which allows the network adapter to reduce copy overhead by accessing application buffers directly. The combination of user-level network access and copy avoidance has a lengthy heritage in research systems spanning two decades [2, 4, 6, 33, 38].

The experiments in Section 7 quantify the improvement in access overhead that DAFS gains from RDMA and transport offload on direct-access NICs.

## 2.3 User-Level File Systems

In addition to overhead reduction, the DAFS protocol leverages user-level networking to enable the network file system structure depicted in the left-hand side of Figure 1. In contrast to traditional kernel-based network file system implementations, as shown in the right side of Figure 1, DAFS file clients may run in user mode as libraries linked directly with applications.

While DAFS also supports kernel-based clients, our work focuses primarily on the properties of the user-level file system structure. A user-level client yields additional modest reductions in overhead by removing system call costs. Perhaps more importantly, it can run on any operating system, with no special kernel support needed other than the NIC driver itself. The client may evolve independently of the operating system, and multiple

client implementations may run on the same system. Most importantly, this structure offers an opportunity to improve integration of file system functions with I/O-intensive applications. In particular, it enables fully asynchronous pipelined file system access, even on systems with inadequate kernel support for asynchronous I/O, and it offers full application control over caching, data movement and prefetching.

It has long been recognized that the kernel policies for file system caching and prefetching are poorly matched to the needs of some important applications [34]. Migrating these OS functions into libraries to allow improved application control and specialization is similar in spirit to the *library operating systems* of Exokernel [21], *protocol service decomposition* for high-speed networking [24], and related approaches. User-level file systems were conceived for the SHRIMP project [4] and the Network-Attached Secure Disks (NASD) project [18]. NFS and other network file system protocols could support user-level clients over an RPC layer incorporating the relevant features of DAFS [7], and we believe that our results and conclusions would apply to such a system.

Earlier work arguing against user-level file systems [39] assumed some form of kernel mediation in the critical I/O path and did not take into account the primary sources of overhead outlined in Section 2.1. However, the user-level structure considered in this paper does have potential disadvantages. It depends on direct-access network hardware, which is not yet widely deployed. Although an application can control caching and prefetching, it does not benefit from the common policies for shared caching and prefetching in the kernel. Thus, in its simplest form, this structure places more burden on the application to manage data movement, and it may be necessary to extend applications to use a new file system API. Section 5 shows how this power and complexity can be encapsulated in prepackaged I/O *adaptation libraries* (depicted in Figure 1) implementing APIs and policies appropriate for a particular class of applications. If the adaptation API has the same syntax and semantics as a pre-existing API, then it is unnecessary to modify the applications themselves (or the operating system).

### 3 DAFS Architecture and Standards

The DAFS specification grew out of the DAFS Collaborative, an industry/academic consortium

led by Network Appliance and Intel, and it is presently undergoing standardization through the Storage Networking Industry Association (SNIA).

The draft standard defines the *DAFS protocol* [14] as a set of request and response formats and their semantics, and a recommended procedural *DAFS API* [13] to access the DAFS service from a client program. Because library-level components may be replaced, client programs may access a DAFS service through any convenient I/O interface. The DAFS API is specified as a recommended interface to promote portability of DAFS client programs. The DAFS API is richer and more complex than common file system APIs including the standard Unix system call interface.

The next section gives an overview of the DAFS architecture and standards, with an emphasis on the transport-related aspects: Sections 3.2 and 3.3 focus on DAFS support for RDMA and asynchronous file I/O respectively.

#### 3.1 DAFS Protocol Summary

The DAFS protocol derives from NFS Version 4 [32] (NFSv4) but diverges from it in several significant ways. DAFS assumes a reliable network transport and offers server-directed command flow-control in a manner similar to block storage protocols such as iSCSI. In contrast to NFSv4, every DAFS operation is a separate request, but DAFS supports request chaining to allow pipelining of dependent requests (e.g., a name *lookup* or *open* followed by file *read*). DAFS protocol headers are organized to preserve alignment of fixed-size fields. DAFS also defines features for reliable session recovery and enhanced locking primitives. To enable the application (or an adaptation layer) to support file caching, DAFS adopts the NFSv4 mechanism for consistent caching based on *open delegations* [1, 14, 32].

The DAFS specification is independent of the underlying transport, but its features depend on direct-access NICs. In addition, some transport-level features (e.g., message flow-control) are defined within the DAFS protocol itself, although they could be viewed as a separate layer below the file service protocol.

#### 3.2 Direct-Access Data Transfer

To benefit from RDMA, DAFS supports *direct* variants of key data transfer operations (*read*, *write*, *readdir*, *getattr*, *setattr*). Direct operations transfer



directly to or from client-provided memory regions using RDMA *read* or *write* operations as described in Section 2.2.

The client must register each memory region with the local kernel before requesting direct I/O on the region. The DAFS API defines primitives to *register* and *unregister* memory regions for direct I/O; the *register* primitive returns a region descriptor to designate the region for direct I/O operations. In current implementations, registration issues a system call to pin buffer regions in physical memory, then loads page translations for the region into a lookup table on the NIC so that it may interpret incoming RDMA directives. To control buffer pinning by a process for direct I/O, the operating system should impose a resource limit similar to that applied in the case of the 4.4BSD *mlock* API [26]. Buffer registration may be encapsulated in an adaptation library.

RDMA operations for direct I/O in the DAFS protocol are always initiated by the server rather than a client. For example, to request a DAFS direct write, the client's write request to the server includes a region token for the buffer containing the data. The server then issues an RDMA *read* to fetch the data from the client, and responds to the DAFS write request after the RDMA completes. This allows the server to manage its buffers and control the order and rate of data transfer [27].

### 3.3 Asynchronous I/O and Prefetching

The DAFS API supports a fully asynchronous interface, enabling clients to pipeline I/O operations and overlap them with application processing. A flexible event notification mechanism delivers asynchronous I/O completions: the client may create an arbitrary number of *completion groups*, specify an arbitrary completion group for each DAFS operation and poll or wait for events on any completion group.

The asynchronous I/O primitives enable event-driven application architectures as an alternative to multithreading. Event-driven application structures are often more efficient and more portable than those based on threads. Asynchronous I/O APIs allow better application control over concurrency, often with lower overhead than synchronous I/O using threads.

Many NFS implementations support a limited form of asynchrony beneath synchronous kernel I/O APIs. Typically, multiple processes (called *I/O daemons* or *nfsiods*) issue blocking requests for sequen-

tial block read-ahead or write-behind. Unfortunately, frequent *nfsiod* context switching adds overhead [2]. The kernel policies only prefetch after a run of sequential reads and may prefetch erroneously if future reads are not sequential.

## 4 DAFS Reference Implementation

We have built prototypes of a user-level DAFS client and a kernel DAFS server implementation for FreeBSD. Both sides of the reference implementation use protocol stubs in a DAFS SDK provided by Network Appliance. The reference implementation currently uses a 1.25 Gb/s Giganet cLAN VI interconnect.

### 4.1 User-level Client

The user-level DAFS client is based on a three-module design, separating transport functions, flow-control and protocol handling. It implements an asynchronous event-driven control core for the DAFS request/response channel protocol. The subset of the DAFS API supported includes direct and asynchronous variants of basic file access and data transfer operations.

The client design allows full asynchrony for single-threaded applications. All requests to the library are non-blocking, unless the caller explicitly requests to wait for a pending completion. The client polls for event completion in the context of application threads, in explicit polling requests and in a standard preamble/epilogue executed on every entry and exit to the library. At these points, it checks for received responses and may also initiate pending sends if permitted by the request flow-control window. Each thread entry into the library advances the work of the client. One drawback of this structure is that pending completions build up on the client receive queues if the application does not enter the library. However, deferring response processing in this case does not interfere with the activity of the client, since it is not collecting its completions or initiating new I/O. A more general approach was recently proposed for asynchronous application-level networking in Exokernel [16].

### 4.2 Kernel Server

The kernel-based DAFS server [25] is a kernel-loadable module for FreeBSD 4.3-RELEASE that implements the complete DAFS specification. Using the VFS/Vnode interface, the server may export

any local file system through DAFS. The kernel-based server also has an event-driven design and takes advantage of efficient hardware support for event notification and delivery in asynchronous network I/O. The server is multithreaded in order to deal with blocking conditions in disk I/O and buffer cache locking.

## 5 Adaptation Libraries

Adaptation libraries are user-level I/O libraries that implement high-level abstractions, caching and prefetching, and insulate applications from the complexity of handling DAFS I/O. Adaptation libraries interpose between the application and the file system interface (e.g., the DAFS API). By introducing versions of the library for each file system API, applications written for the library I/O API can run over user-level or kernel-based file systems, as depicted in Figure 2.

DAFS-based adaptation libraries offer an opportunity to specialize file system functions for classes of applications. For example, file caching at the application level offers three potential benefits. First, the application can access a user-level cache with lower overhead than a kernel-based cache accessed through the system call interface. Second, the client can efficiently use application-specific fetch and replacement policies. Third, in cases where caching is an essential function of the adaptation library, a user-level file system avoids the problem of *double caching*, in which data is cached redundantly in the kernel-level and user-level caches.

One problem with user-level caching is that the kernel virtual memory system may evict cached pages if the client cache consumes more memory than the kernel allocates to it. For this reason, each of these adaptation libraries either pre-registers its cache (as described in Section 3.2) or configures it to a “safe” size. A second problem is that user-level caches are not easily shared across multiple uncooperating applications, but the need for this sharing is less common in high-performance domains.

To illustrate the role of adaptation libraries, we consider two examples that we enhanced for use with DAFS: TPIE and Berkeley DB.

### 5.1 TPIE

TPIE [37] (Transparent Parallel I/O Environment) is a toolkit for *external memory* (EM) algorithms. EM algorithms are structured to handle

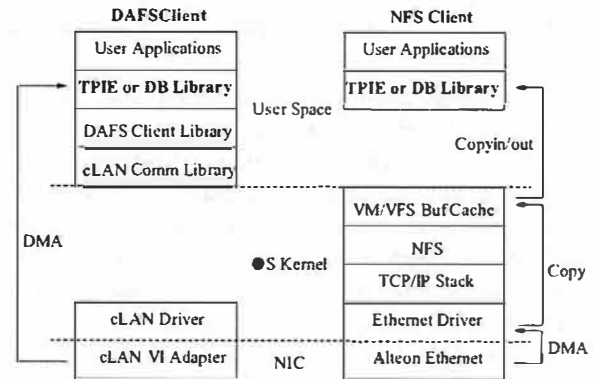


Figure 2: Adaptation Libraries can benefit from user-level clients without modifying applications.

massive data problems efficiently by minimizing the number of I/O operations. They enhance locality of data accesses by performing I/O in large blocks and maximizing the useful computation on each block while it is in memory. The TPIE toolkit supports a range of applications including Geographic Information System (GIS) analysis programs for massive terrain grids [3].

To support EM algorithms, TPIE implements a dataflow-like streaming paradigm on collections of fixed-size records. It provides abstract stream types with high-level interfaces to “push” streams of data records through application-defined record operators. A pluggable Block Transfer Engine (BTE) manages transfer buffering and provides an interface to the underlying storage system. We introduced a new BTE for the DAFS interface, allowing us to run TPIE applications over DAFS without modification. The BTE does read-ahead and write-behind on data streams using DAFS asynchronous primitives, and handles the details of block clustering and memory registration for direct I/O.

### 5.2 Berkeley DB

Berkeley DB (*db*) [28] is an open-source embedded database system that provides library support for concurrent storage and retrieval of key/value pairs. *Db* manages its own buffering and caching, independent of any caching at the underlying file system buffer cache. *Db* can be configured to use a specific page size (the unit of locking and I/O, usually 8KB) and buffer pool size. Running *db* over DAFS avoids double caching and bypasses the standard file system prefetching heuristics, which may degrade performance for common *db* access patterns. Section 7.4 shows the importance of these effects for *db*



performance.

## 6 Low-Overhead Kernel-Based NFS

The DAFS approach is one of several alternatives to improving access performance for network storage. In this section we consider a prominent competing structure as a basis for the empirical comparisons in Section 7. This approach enhances a kernel-based NFS client to reduce overhead for protocol processing and/or data movement. While this does not reduce system-call costs, there is no need to modify existing applications or even to re-link them if the kernel API is preserved. However, it does require new kernel support, which is a barrier to fast and reliable deployment. Like DAFS, meaningful NFS enhancements of this sort also rely on new support in the NIC.

The most general form of copy avoidance for file services uses header splitting and page flipping, variants of which have been used with TCP/IP protocols for more than a decade (e.g., [5, 8, 11, 35]). To illustrate, we briefly describe FreeBSD enhancements to extend copy avoidance to *read* and *write* operations in NFS. Most NFS implementations send data directly from the kernel file cache without making a copy, so a client initiating a *write* and a server responding to a *read* can avoid copies. We focus on the case of a client receiving a *read* response containing a block of data to be placed in the file cache. The key challenge is to arrange for the NIC to deposit the data payload—the file block—page-aligned in one or more physical page frames. These pages may then be inserted into the file cache by reference, rather than by copying. It is then straightforward to deliver the data to a user process by remapping pages rather than by physical copy, but only if the user's buffers are page-grained and suitably aligned. This also assumes that the file system block size is an integral multiple of the page size.

To do this, the NIC first strips off any transport headers and the NFS header from each message and places the data in a separate page-aligned buffer (*header splitting*). Note that if the network MTU is smaller than the hardware page size, then the transfer of a page of data is spread across multiple packets, which can arrive at the receiver out-of-order and/or interspersed with packets from other flows. In order to pack the data contiguously into pages, the NIC must do significant protocol processing for NFS and its transport to decode the incoming packets. NFS complicates this processing with variable-length headers.

We modified the firmware for Alteon Tigon-II Gigabit Ethernet adapters to perform header splitting for NFS *read response* messages. This is sufficient to implement a zero-copy NFS client. Our modifications apply only when the transport is UDP/IP and the network is configured for Jumbo Frames, which allow NFS to exchange data in units of pages. To allow larger block sizes, we altered IP fragmentation code in the kernel to avoid splitting page buffers across fragments of large UDP packets. Together with other associated kernel support in the file cache and VM system, this allows zero-copy data exchange with NFS block-transfer sizes up to 32KB. Large NFS transfer sizes can reduce overhead for bulk data transfer by limiting the number of trips through the NFS protocol stack; this also reduces transport overheads on networks that allow large packets.

While this is not a general solution, it allows us to assess the performance potential of optimizing a kernel-based file system rather than adopting a direct-access user-level file system architecture like DAFS. It also approximates the performance achievable with a kernel-based DAFS client, or an NFS implementation over VI or some other RDMA-capable network interface. As a practical matter, the RDMA approach embraced in DAFS is a more promising alternative to low-overhead NFS. Note that page flipping NFS is much more difficult over TCP, because the NIC must buffer and reassemble the TCP stream to locate NFS headers appearing at arbitrary offsets in the stream. This is possible in NICs implementing a TCP offload engine, but impractical in conventional NICs such as the Tigon-II.

## 7 Experimental Results

This section presents performance results from a range of benchmarks over our DAFS reference implementation and two kernel-based NFS configurations. The goal of the analysis is to quantify the effects of the various architectural features we have discussed and understand how they interact with properties of the workload.

Our system configuration consists of Pentium III 800MHz clients and servers with the ServerWorks LE chipset, equipped with 256MB-1GB of SDRAM on a 133 MHz memory bus. Disks are 9GB 10000 RPM Seagate Cheetahs on a 64-bit/33 MHz PCI bus. All systems run patched versions of FreeBSD 4.3-RELEASE. DAFS uses VI over Gigaset cLAN 1000 adapters. NFS uses UDP/IP over Gigabit Ethernet, with Alteon Tigon-II adapters.

Table 1: Baseline Network Performance.

	VI/cLAN	UDP/Tigon-II
Latency	30 $\mu$ s	132 $\mu$ s
Bandwidth	113 MB/s	120 MB/s

Table 1 shows the raw one-byte roundtrip latency and bandwidth characteristics of these networks. The Tigon-II has a higher latency partly due to the datapath crossing the kernel UDP/IP stack. The bandwidths are comparable, but not identical. In order to best compare the systems, we present performance results in Sections 7.1 and 7.2 normalized to the maximum bandwidth achievable on the particular technology.

NFS clients and servers exchange data in units of 4KB, 8KB, 16KB or 32KB (the NFS block I/O transfer size is set at mount time), sent in fragmented UDP packets over 9000-byte MTU Jumbo Ethernet frames to minimize data transfer overheads. Checksum offloading on the Tigon-II is enabled, minimizing checksum overheads. Interrupt coalescing on the Tigon-II was set as high as possible without degrading the minimum one-way latency of about 66 $\mu$ s for one-byte messages. NFS client (*nfsiod*) and server (*nfsd*) concurrency was tuned for best performance in all cases.

For NFS experiments with copy avoidance (*NFS-nocopy*), we modified the Tigon-II firmware, IP fragmentation code, file cache code, VM system and Tigon-II driver for NFS/UDP header splitting and page remapping as described in Section 6. This configuration is the state of the art for low-overhead data transfer over NFS. Experiments with the standard NFS implementation (*NFS*) use the standard Tigon-II driver and vendor firmware.

## 7.1 Bandwidth and Overhead

We first explore the bandwidth and client CPU overhead for reads with and without read-ahead. These experiments use a 1GB server cache prewarmed with a 768MB dataset. For this experiment, we factor out client caching as the NFS client cache is too small to be effective and the DAFS client does not cache. Thus these experiments are designed to stress the network data transfer. These results are representative of workloads with sequential I/O on large disk arrays or asynchronous random-access loads on servers with sufficient disk arms to deliver data to the client at net-

work speed. None of the architectural features discussed yield much benefit for workloads and servers that are disk-bound, as they have no effect on the remote file system or disk system performance, as shown in Section 7.4.

The application's request size for each file I/O request is denoted *block size* in the figures. The ideal block size depends on the nature of the application; large blocks are preferable for applications that do long sequential reads, such as streaming multimedia, and smaller blocks are useful for nonsequential applications, such as databases.

In the read-ahead (sequential) configurations, the DAFS client uses the asynchronous I/O API (Section 3.3), and NFS has kernel-based sequential read-ahead enabled. For the experiments without read-ahead (random access), we tuned NFS for best-case performance at each request size (block size). For request sizes up to 32K, NFS is configured for a matching NFS transfer size, with read-ahead disabled. This avoids unnecessary data transfer or false prefetching. For larger request sizes we used an NFS transfer size of 32K and implicit read-ahead up to the block size. One benefit of the user-level file system structure is that it allows clients to select transfer size and read-ahead policy on a per-application basis. All DAFS experiments use a transfer size equal to the request block size.

**Random access reads with read-ahead disabled.** The left graphs in Figure 3 and Figure 4 reports bandwidth and CPU utilization, respectively, for random block reads with read-ahead disabled. All configurations achieve progressively higher bandwidths with increasing block size, since the wire is idle between requests. The key observation here is that with small request sizes the DAFS configuration outperforms both NFS implementations by a factor of two to three. This is due to lower operation latency, stemming primarily from the lower network latency (see Table 1) but also from the lower per-I/O overhead. This lower overhead results from the transport protocol offload to the direct-access NIC, including reduced system-call and interrupt overhead possible with user-level networking.

With large request sizes, transfer time dominates. NFS peaks at less than one-half the link bandwidth (about 60 MB/s), limited by memory copying overhead that saturates the client CPU. DAFS achieves wire speed using RDMA with low client CPU utilization, since the CPU is not involved in the RDMA transfers. NFS-nocopy eliminates the copy overhead with page flipping; it also

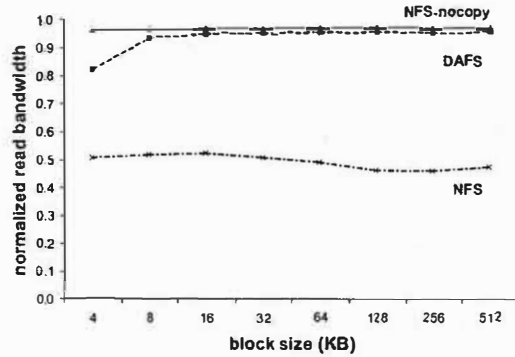
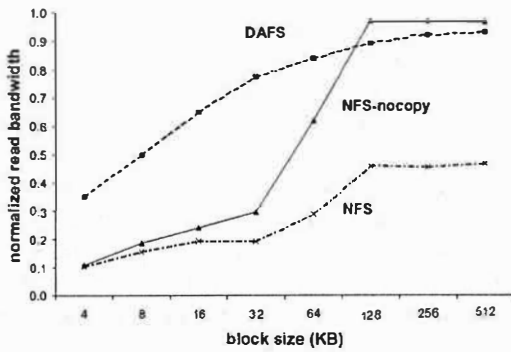


Figure 3: Read bandwidth without (left) and with (right) read-ahead.

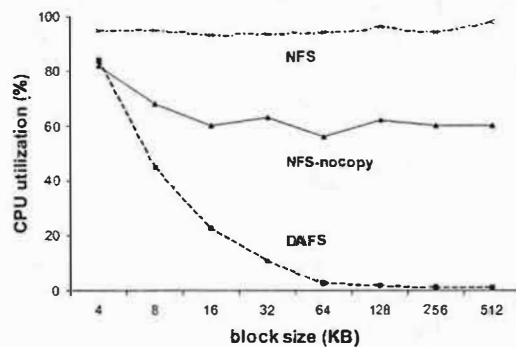
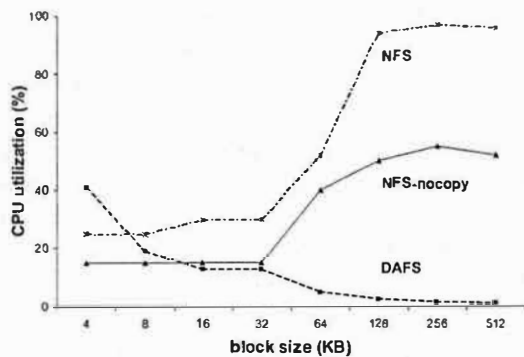


Figure 4: Read client CPU utilization (%) without (left) and with (right) read-ahead.

approaches wire speed for large block sizes, but the overhead for page flipping and kernel protocol code consumes 50% of the client CPU at peak bandwidth.

**Sequential reads with read-ahead.** The right graphs in Figure 3 and Figure 4 report bandwidths and CPU utilization for a similar experiment using sequential reads with read-ahead enabled. In this case, all configurations reach their peak bandwidth even with small block sizes. Since the bandwidths are roughly constant, the CPU utilization figures highlight the differences in the protocols. Again, when NFS peaks, the client CPU is saturated; the overhead is relatively insensitive to block size because it is dominated by copying overhead.

Both NFS-nocopy and DAFS avoid the byte copy overhead and exhibit lower CPU utilization than NFS. However, while CPU utilization drops off with increasing block size, this drop is significant for DAFS, but noticeably less so for NFS-nocopy. The NFS-nocopy overhead is dominated by page flipping and transport protocol costs, both of which are insensitive to block size changes beyond

the page size and network MTU size. In contrast, the DAFS overhead is dominated by request initiation and response handling costs in the file system client code, since the NIC handles data transport using RDMA. Therefore, as the number of requests drops with the increasing block size, the client CPU utilization drops as well.

The following experiments illustrate the importance of these factors for application performance.

## 7.2 TPIE Merge

This experiment combines raw sequential I/O performance, including writes, with varying amounts of application processing. As in the previous experiment, we configure the system to stress client data-transfer overheads, which dominate when the server has adequate CPU and I/O bandwidth for the application. In this case the I/O load is spread across two servers using 600MB memory-based file systems. Performance is limited by the client CPU rather than the server CPU, network, or disk I/O.

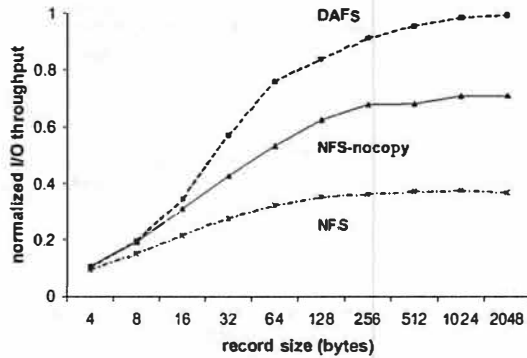


Figure 5: TPIE Merge throughput for  $n = 8$ .

The benchmark is a TPIE-based sequential record *Merge* program, which combines  $n$  sorted input files of  $x$   $y$ -byte records, each with a fixed-size key (an integer), into one sorted output file. Performance is reported as total throughput:

$$\frac{2 \cdot n \cdot x \cdot y}{t} \quad \frac{(\text{bytes})}{(\text{sec})}$$

This experiment shows the effect of low-overhead network I/O on real application performance, since the merge processing competes with I/O overhead for client CPU cycles. Varying the merge order  $n$  and/or record size  $y$  allows us to control the amount of CPU work the application performs per block of data. CPU cost per record (key comparisons) increases logarithmically with  $n$ ; CPU cost per byte decreases linearly with record size. This is because larger records amortize the comparison cost across a larger number of bytes, and there are fewer records per block.

We ran the *Merge* program over two variants of the TPIE library, as described in Section 5. One variant (TPIE/DAFS) is linked with the user-level DAFS client and accesses the servers using DAFS. In this variant, TPIE manages the streaming using asynchronous I/O, with zero-copy reads using RDMA and zero-copy writes using inline DAFS writes over the cLAN's scatter/gather messaging (cLAN does not support DAFS writes using RDMA). The second variant (TPIE/NFS) is configured to use the standard kernel file system interface to access the servers over NFS. For TPIE/NFS, we ran experiments using both standard NFS and NFS-nocopy configurations, as in the previous section. For the NFS configurations, we tuned read-ahead and write-behind concurrency (*nfsiods*) for the best performance in all cases. The TPIE I/O request size was fixed to 32KB.

Figure 5 shows normalized merge throughput

results; these are averages over ten runs with a variance of less than 2% of the average. The merge order is  $n=8$ . The record size varies on the  $x$ -axis, showing the effect of changing the CPU demands of the application. The results show that the lower overhead of the DAFS client (noted in the previous section) leaves more CPU and memory cycles free for the application at a given I/O rate, resulting in higher merge throughputs. Note that the presence of application processing accentuates the gap relative to the raw-bandwidth tests in the previous subsection. It is easy to see that when the client CPU is saturated, the merge throughput is inversely proportional to the total processing time per block, i.e., the sum of the total per-block I/O overhead and application processing time.

For example, on the right side of the graph, where the application has the highest I/O demand (due to larger records) and hence the highest I/O overhead, DAFS outperforms NFS-nocopy by as much as 40%, as the NFS-nocopy client consumes up to 60% of its CPU in the kernel executing protocol code and managing I/O. Performance of the NFS configuration is further limited by memory copying through the system-call interface in writes.

An important point from Figure 5 is that the relative benefit of the low-overhead DAFS configuration is insignificant when the application is compute-bound. As application processing time per block diminishes (from left to right in Figure 5), reducing I/O overhead yields progressively higher returns because the I/O overhead is a progressively larger share of total CPU time.

### 7.3 PostMark

PostMark [22] is a synthetic benchmark aimed at measuring file system performance over a workload composed of many short-lived, relatively small files. Such a workload is typical of mail and net-news servers used by Internet Service Providers. PostMark workloads are characterized by a mix of metadata-intensive operations. The benchmark begins by creating a pool of files with random sizes within a specified range. The number of files, as well as upper and lower bounds for file sizes, are configurable. After creating the files, a sequence of transactions is performed. These transactions are chosen randomly from a file creation or deletion operation paired with a file read or write. A file creation operation creates and writes random text to a file. File deletion removes a random file from the active set. File read reads a random file in its entirety

and file write appends a random amount of data to a randomly chosen file. In this section, we consider DAFS as a high-performance alternative to NFS for deployment in mail and netnews servers [9, 10].

We compare PostMark performance over DAFS to NFS-nocopy. We tune NFS-nocopy to exhibit best-case performance for the average file size used each time. For file sizes less than or equal to 32K, NFS uses a block size equal to the file size (to avoid read-ahead). For larger file sizes, NFS uses 32K blocks, and read-ahead is adjusted according to the file size. Read buffers are page-aligned to enable page remapping in delivering data to the user process. In all cases an FFS file system is exported using soft updates [17] to eliminate synchronous disk I/O for metadata updates.

Our NFS-nocopy implementation is based on NFS Version 3 [30]. It uses write-behind for file data: writes are delayed or asynchronous depending on the size of the data written. On close, the NFS client flushes dirty buffers to server memory and waits for flushing to complete but does not commit them to server disk. NFS Version 3 *open-to-close consistency* dictates that cached file blocks be re-validated with the server each time the file is opened<sup>1</sup>.

With DAFS, the client does not cache and all I/O requests go to the server. Writes are not synchronously committed to disk, offering a data reliability similar to that provided by NFS. DAFS inlines data with the write RPC request (since the cLAN cannot support server-initiated RDMA read operations required for direct file writes) but uses direct I/O for file reads. In both cases, the client waits for the RPC response.

A key factor that determines PostMark performance is the cost of metadata operations (e.g., open, close, create, delete). Enabling soft updates on the server-exported filesystem decouples metadata updates from the server disk I/O system. This makes the client metadata operation cost sensitive to the client-server RPC as well as to the metadata update on the server filesystem. As the number of files per directory increases, the update time on the server dominates because FFS performs a linear time lookup. Other factors affecting performance are client caching and network I/O.

In order to better understand the PostMark

<sup>1</sup>In the NFS implementation we used, writing to a file before closing it originally resulted in an invalidation of cached blocks in the next open of that file, as the client could not tell who was responsible for the last write. We modified our implementation to avoid this problem.

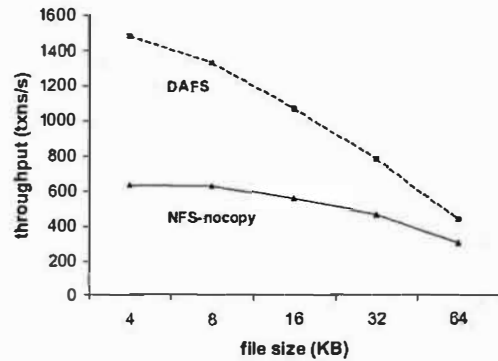


Figure 6: **PostMark.** Effect of I/O boundedness.

performance, we measured the latency of the micro-operations involved in each PostMark transaction. As we saw in Table 1, the Tigon-II has significantly higher roundtrip latency than the cLAN. As a result, there is a similar difference in the null-RPC cost which makes the cLAN RPC three times faster than Tigon-II RPC ( $47\mu s$  vs.  $154\mu s$ ). In addition, all DAFS file operations translate into a single RPC to the server. With NFS, a file create or remove RPC is preceded by a second RPC to get the attributes of the directory that contains the file. Similarly, a file open requires an RPC to validate the file's cached blocks. The combination of more expensive and more frequent RPCs introduces a significant performance differential between the two systems.

This experiment measures the effect of increasing I/O boundedness on performance. We increase the average file size from 4KB to 64KB, maintaining a small number of files (about 150) to minimize the server lookup time. Each run performs 30,000 transactions, each of which is a create or delete paired with a read operation. We report the average of ten runs, which have a variance under 10% of the average. The client and server have 256MB and 1GB of memory, respectively. At small file sizes, the increased cost of metadata operations for NFS-nocopy dominates its performance. As the I/O boundedness of the workload increases, DAFS performance becomes dominated by network I/O transfers and drops linearly. For large file sizes, reads under NFS-nocopy benefit from caching, but writes still have to go to the server to retain open-to-close semantics.

This experiment shows that DAFS outperforms NFS-nocopy by more than a factor of two for small files due largely to the lower latency of metadata operations on the cLAN network. Part of this difference could be alleviated by improvements in networking technology. However, the difference



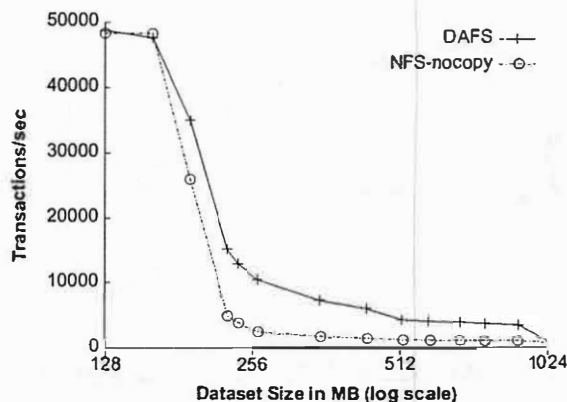


Figure 7: Berkeley DB. Effect of double caching and remote memory access performance.

in the number of RPCs between DAFS and NFS is fundamental to the protocols. For larger files, the NFS-nocopy benefit from client caching is limited by its consistency model that requires waiting on outstanding asynchronous writes on file close.

## 7.4 Berkeley DB

In this experiment, we use a synthetic workload composed of read-only transactions, each accessing one small record uniformly at random from a B-tree to compare *db* performance over DAFS to NFS-nocopy. The workload is single-threaded and read-only, so there is no logging or locking. In all experiments, after warming the *db* cache we performed a sequence of read transactions long enough to ensure that each record in the database is touched twice on average. The results report throughput in transactions per second. The unit of I/O is a 16KB block.

We vary the size of the *db* working set in order to change the bottleneck from local memory, to remote memory, to remote disk I/O. We compare a DAFS *db* client to an NFS-nocopy *db* client both running on a machine with 256MB of physical memory. In both cases the server runs on a machine with 1GB of memory. Since we did not expect read-ahead to help in the random access pattern considered here, we disable read-ahead for NFS-nocopy and use a transfer size of 16K. The *db* user-level cache size is set to the amount of physical memory expected to be available for allocation by the user process. The DAFS client uses about 36MB for communication buffers and statically-sized structures leaving about 190MB to the *db* cache. To facilitate comparison between the systems, we con-

figure the cache identically for NFS-nocopy.

Figure 7 reports throughput with warm *db* and server cache. In NFS-nocopy, reading through the file system cache creates competition for physical memory between the user-level and file system caches (Section 5). For database sizes up to the size of the *db* cache, the user-level cache is able to progressively use more physical memory during the warming period, as network I/O diminishes. Performance is determined by local memory access as *db* eventually satisfies requests entirely from the local cache.

Once the database size exceeds the client cache, performance degrades as both systems start accessing remote memory. NFS-nocopy performance degrades more sharply due to two effects. First, the double caching effect creating competition for physical memory between the user-level and file system caches is now persistent due to increased network I/O demands. As a result, the filesystem cache grows and user-level cache memory is paged out to disk causing future page faults. Second, since the *db* API issues unaligned page reads from the file system, NFS-nocopy cannot use page remapping to deliver data to the user process. The DAFS client avoids these effects by maintaining a single client cache and doing direct block reads into *db* buffers. For database sizes larger than 1GB that cannot fit in the server cache, both systems are disk I/O bound on the server.

## 8 Conclusions

This paper explores the key architectural features of the Direct Access File System (DAFS), a new architecture and protocol for network-attached storage over direct-access transport networks. DAFS or other approaches that exploit such networks and user-level file systems have the potential to close the performance gap between full-featured network file services and network storage based on block access models.

The contribution of our work is to characterize the issues that determine the effects of DAFS on application performance, and to quantify these effects using experimental results from a public DAFS reference implementation for an open-source Unix system (FreeBSD). For comparison, we report results from an experimental zero-copy NFS implementation. This allows us to evaluate the sources of the performance improvements from DAFS (e.g., copy overhead vs. protocol overhead) and the alternatives for achieving those benefits without DAFS.

DAFS offers significant overhead reductions for high-speed data transfer. These improvements result primarily from direct access and RDMA, the cornerstones of the DAFS design, and secondarily from the effect of transport offload to the network adapter. These benefits can yield significant application improvements. In particular, DAFS delivers the strongest benefit for balanced workloads in which application processing saturates the CPU when I/O occurs at network speed. In such a scenario, DAFS improves application performance by up to 40% over NFS-nocopy for TPIE *Merge*. However, many workload factors can undermine these benefits. Direct-access transfer yields little benefit with servers that are disk-limited, or with workloads that are heavily compute-bound.

Our results also show that I/O adaptation libraries can obtain benefits from the DAFS user-level client architecture, without the need to port applications to a new (DAFS) API. Most importantly, adaptation libraries can leverage the additional control over concurrency (asynchrony), data movement, buffering, prefetching and caching in application-specific ways, without burdening applications. This creates an opportunity to address longstanding problems related to the integration of the application and file system for high-performance applications.

## 9 Acknowledgments

This research was supported by the National Science Foundation (through grants CCR-00-82912, EIA-9972879, and EIA-9870728), Network Appliance, and IBM. We thank our shepherd Greg Ganger and the anonymous reviewers for their valuable comments.

## 10 Software Availability

The DAFS and NFS-nocopy implementations used in this paper are available from <http://www.eecs.harvard.edu/vino/fs-perf/dafs> and <http://www.cs.duke.edu/ari/dafs>.

## References

- [1] S. Addetia. User-level Client-side Caching for DAFS. Technical report, Harvard University TR-14-01, March 2002.
- [2] D. Anderson, J. S. Chase, S. Gadde, A. Gallatin, and K. Yocum. Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet. In *Proc. of Usenix Technical Conference, New Orleans, LA*, June 1998.
- [3] L. Arge, J. S. Chase, L. Toma, J. S. Vitter, R. Wickremesinghe, P. N. Halpin, and D. Urban. Flow computation on massive grids. In *Proc. of ACM-GIS, ACM Symposium on Advances in Geographic Information Systems*, November 2001.
- [4] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, and E. Felten. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. of 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.
- [5] J. Brustoloni. Interoperation of Copy Avoidance in Network and File I/O. In *Proc. of 18th IEEE Conference on Computer Communications (INFOCOM'99)*, New York, NY, March 1999.
- [6] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proc. of Second Symposium on Operating Systems Design and Implementation*, October 1996.
- [7] B. Callaghan. NFS over RDMA. Work-in-Progress Presentation. *USENIX File Access and Storage Symposium, Monterey, CA*, January 2002.
- [8] J. S. Chase, A. J. Gallatin, and K. G. Yocum. End system optimizations for high-speed TCP. *IEEE Communications, Special Issue on TCP Performance in Future Networking Environments*, 39(4):68–74, April 2001.
- [9] N. Christenson, D. Beckermeier, and T. Baker. A Scalable News Architecture on a Single Pool. *login*, 22(3):41–45, December 1997.
- [10] N. Christenson, T. Bosserman, and D. Beckermeier. A Highly Scalable Electronic Mail Service Using Open Systems. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [11] H. J. Chu. Zero-Copy TCP in Solaris. In *Proc. of USENIX Technical Conference, San Diego, CA*, January 1996.
- [12] Compaq, Intel, Microsoft. *Virtual Interface Architecture Specification, Version 1.0*, December 1997.
- [13] DAFS Collaborative. *DAFS API, Version 1.0*, November 2001.
- [14] DAFS Collaborative. *Direct Access File System Protocol, Version 1.0*, September 2001. <http://www.dafscollaborative.org>.
- [15] C. Dalton, G. Watson, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner: A Network-Independent Card Provides Architectural Support for High-Performance Protocols. *IEEE Network*, pages 36–43, July 1993.
- [16] G. Ganger, D. Engler, M. F. Kaashoek, H. Briceno, R. Hunt, and T. Pinckney. Fast and Flexible

- Application-Level Networking on Exokernel Systems. *ACM Transactions on Computer Systems*, 20(1):49–83, February 2002.
- [17] G. Ganger and Y. Patt. Metadata Update Performance in File Systems. In *Proc. of USENIX Symposium on Operating System Design and Implementation*, pages 49–60, November 1994.
  - [18] G. Gibson, D. Nagle, K. Amiri, J. Buttler, F. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proc. of 8th Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
  - [19] P. Joubert, R. King, R. Neves, M. Russinovich, and J. Tracey. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. In *Proc. of USENIX Technical Conference, Boston, MA*, July 2001.
  - [20] C. Jurgens. Fibre Channel: A Connection to the Future. *IEEE Computer*, 28(8):88–90, August 1995.
  - [21] M. F. Kaashoek, D. Engler, G. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility in Exokernel Systems. In *Proc. of 16th Symposium on Operating Systems Principles*, October 1997.
  - [22] J. Katcher. PostMark: A New File System Benchmark. Technical report, Network Appliance TR-3022, October 1997.
  - [23] E. Lee and C. Thekkath. Petal: Distributed Virtual Disks. In *Proc. of 7th International Conference on Architectural Support for Programming Languages and Operating Systems (Cambridge, Mass., Oct.)*. ACM Press, New York, pages 84–92, October 1996.
  - [24] C. Maeda and B. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proc. of Symposium on Operating Systems Principles*, pages 244–255, 1993.
  - [25] K. Magoutis. Design and Implementation of a Direct Access File System (DAFS) Kernel Server for FreeBSD. In *Proc. of USENIX BSDCon Conference, San Francisco, CA*, February 2002.
  - [26] M. K. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.
  - [27] D. Nagle, G. Ganger, J. Butler, G. Goodson, and C. Sabol. Network Support for Network-Attached Storage. In *Proc. of Hot Interconnects, Stanford, CA*, August 1999.
  - [28] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proc. of USENIX Technical Conference, FREENIX Track, Monterey, CA*, June 1999.
  - [29] V. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching Scheme. In *Proc. of Third USENIX Symposium on Operating System Design and Implementation, New Orleans, LA*, February 1999.
  - [30] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. In *Proc. of USENIX Technical Conference, Boston, MA*, June 1994.
  - [31] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. of Symposium on Operating System Design and Implementation, Seattle, WA*, October 1996.
  - [32] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. RFC 3010, December 2000.
  - [33] A. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4), pages 246–260, April 1982.
  - [34] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.
  - [35] M. Thadani and Y. Khalidi. An Efficient Zero-copy I/O Framework for UNIX. Technical report, SMLI TR95-39, Sun Microsystems Lab, Inc., May 1995.
  - [36] C. Thekkath, T. Mann, and E. Lee. Frangipani: A Scalable Distributed File System. In *Proc. of 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.
  - [37] D. E. Vengroff and J. S. Vitter. I/O-efficient computation: The TPIE approach. In *Proc. of the Goddard Conference on Mass Storage Systems and Technologies*, NASA Conference Publication 3340, Volume II, pages 553–570, College Park, MD, September 1996.
  - [38] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.
  - [39] B. Welch. The File System Belongs in the Kernel. In *Proc. of Second USENIX Mach Symposium*, November 1991.



# Conquest: Better Performance Through A Disk/Persistent-RAM Hybrid File System

An-I A. Wang, Peter Reiher, and Gerald J. Popek<sup>\*</sup>  
Computer Science Department  
University of California, Los Angeles  
{awang, reiher, popek}@fmg.cs.ucla.edu

Geoffrey H. Kuenning  
Computer Science Department  
Harvey Mudd College  
geoff@cs.hmc.edu

## Abstract

*The rapidly declining cost of persistent RAM technologies prompts the question of when, not whether, such memory will become the preferred storage medium for many computers. Conquest is a file system that provides a transition from disk to persistent RAM as the primary storage medium. Conquest provides two specialized and simplified data paths to in-core and on-disk storage, and Conquest realizes most of the benefits of persistent RAM at a fractional cost of a RAM-only solution. As of October 2001, Conquest can be used effectively for a hardware cost of under \$200.*

*We compare Conquest's performance to ext2, reiserfs, SGI XFS, and ramfs, using popular benchmarks. Our measurements show that Conquest incurs little overhead compared to ramfs. Compared to the disk-based file systems, Conquest achieves 24% to 1900% faster memory performance, and 43% to 96% faster performance when exercising both memory and disk.*

## 1 Introduction

For over 25 years, long-term storage has been dominated by rotating magnetic media. At the beginning of the disk era, tapes were still widely used for online storage; today, they are almost exclusively used for backup despite still being cheaper than disks. The reasons are both price threshold and performance: although disks are more expensive, they are cheap enough for common use, and their performance is vastly superior.

Today, the rapidly dropping price of RAM suggests that a similar transition may soon take place, with all-electronic technologies gradually replacing disk storage. This transition is already happening in portable devices such as cameras, PDAs, and MP3 players. Because rotational delays are not relevant to

persistent RAM storage, it is appropriate to consider whether existing file system designs are suitable in this new environment.

The *Conquest* file system is designed to address these questions and to smooth the transition from disk-based to persistent-RAM-based storage. Unlike other memory file systems [21, 10, 43], *Conquest* provides an incremental solution that assumes more file system responsibility in-core as memory prices decline. Unlike HeRMES [25], which deploys a relatively modest amount of persistent RAM to alleviate disk traffic, *Conquest* assumes an abundance of RAM to perform most file system functions. In essence, *Conquest* provides two specialized and simplified data paths to in-core and on-disk storage. *Conquest* achieves most of the benefits of persistent RAM without the full cost of RAM-only solutions. As persistent RAM becomes cheaply abundant, *Conquest* can realize more additional benefits incrementally.

## 2 Alternatives to Conquest

Given the promise of using increasingly cheap memory to improve file systems performance, it would be desirable to do so as simply as possible. However, the obvious simple methods for gaining such benefits fail to take complete advantage of the new possibilities. In many cases, extensions to the simple methods can give results similar to our approach, but to make these extensions, so much complexity must be added that they are no longer attractive alternatives to the *Conquest* approach.

In this section, we will discuss the limitations of these alternatives. Some do not provide the expected performance gains, while others do not provide a complete solution to the problem of storing arbitrary amounts of data persistently, reliably, and conveniently. Rather than adding the complications necessary to fix these approaches, it is better to start the design with a clean slate.

<sup>\*</sup> Gerald Popek is also associated with United On-Line.

## 2.1 Caching

One alternative to a hybrid RAM-based file system like *Conquest* is instead to take advantage of the existing file buffer cache. Given that a computer has an ample amount of RAM, why not just allocate that RAM to a buffer cache, rather than dedicating it to a file storage system? This approach seems especially appropriate because the buffer cache tends to populate itself with the most frequently referenced files, rather than wasting space on files that have been untouched for lengthy periods.

However, using the buffer cache has several drawbacks. Roselli et al., [34] showed that caching often experiences diminishing marginal returns as the size of cache grows larger. They also found that caches could experience miss rates as high as 10% for some workloads, which is enough to reduce performance significantly.

Another challenge is handling cache pollution, which can have a variety of causes—reading large files, buffering asynchronous writes, daily backups, global searches, disk maintenance utilities, etc. This problem led to remedies such as LFU buffer replacement for large files or attempts to reduce cache-miss latency by modifying compilers [39], placing the burden on the programmer [31], or constructing user behavior-analysis mechanisms within the kernel [15, 19].

Caches also make it difficult to maintain data consistency between memory and disk. A classic example is metadata commits, which are synchronous in most file systems. Asynchronous solutions do exist, but at the cost of code complexity [12, 38].

Moving data between disk and memory can involve remarkably complex management. For example, moving file data from disk to memory involves locating the metadata, scheduling the metadata transfer to memory, translating the metadata into runtime form, locating data and perhaps additional metadata, scheduling the data transfer, and reading the next data block ahead of time.

*Conquest* fundamentally differs from caching by not treating memory as a scarce resource. Instead, *Conquest* anticipates the abundance of cheap persistent RAM and uses disk to store the data well suited to disk characteristics. We can then achieve simpler disk optimizations by narrowing the range of access patterns and characteristics anticipated by the file system.

## 2.2 RAM Drives and RAM File Systems

Many computer scientists are so used to disk storage that we sometimes forget that persistence is not automatic. In addition to the storage medium, persistence also requires a protocol for storing and retrieving the information from the persistent medium,

so that a file system can survive reboots. While persistent RAM provides nonvolatility of memory content, the file system and the memory manager also need to know how to take advantage of the storage medium.

Most RAM disk drivers operate by emulating a physical disk drive. Although there is a file system protocol for storing and retrieving the in-memory information, there is no protocol to recover the associated memory states. Given that the existing memory manager is not aware of RAM drives, isolating these memory states for persistence can be nontrivial.

RAM file systems under Linux and BSD [21] use the IO caching infrastructure provided by VFS to store both metadata and data in various temporary caches directly. Since the memory manager is unaware of RAM file systems, neither the file system nor the memory states survive reboots without significant modifications to the existing memory manager.

Both RAM drives and RAM file systems also incur unnecessary disk-related overhead. For RAM drives, existing file systems, tuned for disk, are installed on the emulated drive without regard for the absence of the mechanical limitations of disks. For example, access to RAM drives is done in blocks, and the file system will still waste effort attempting to place files in "cylinder groups" even though cylinders and block boundaries no longer exist. Although RAM file systems have eliminated some disk-related complexities, many RAM file systems rely on VFS and its generic storage access routines; many built-in mechanisms such as readahead and buffer-cache reflect assumptions that the underlying storage medium is slower than memory.

In addition, both RAM drives and RAM file systems limit the size of the files they can store to the size of main memory. These restrictions have limited the use of RAM disks to caching and temporary file systems. To move to a general-purpose persistent-RAM file system, we need a substantially new design.

## 2.3 Disk Emulators

Some manufacturers advocate RAM-based disk emulators for specialty applications [44]. These emulators generally plug into a standard SCSI or similar IO port, and look exactly like a disk drive to the CPU. Although they provide a convenient solution to those who need an instant speedup, and they do not suffer the persistence problem of RAM disks, they again are an interim solution that does not address the underlying problem and does not take advantage of the unique benefits of RAM. In addition, standard IO interfaces force the emulators to operate through inadequate access methods and low-bandwidth cables, greatly limiting the utility of this option [33] as something other than a stopgap measure.

## 2.4 Ad Hoc Approaches

There are also a number of less structured approaches to using existing tools to exploit the abundance of RAM. For example, one could achieve persistence by manually transferring files into *ramfs* at boot time and preserving them again before shutdown. However, this method would drastically limit the total file system size.

Another option is to attempt to manage RAM space by using a background daemon to stage files to a disk partition. Although this could be made to work, it would require significant additional complexity to maintain the single name space provided by *Conquest* and to preserve the semantics of symbolic and hard links when moving files between storage media.

## 3 Conquest File System Design

Our initial design assumes the popular single-user desktop environment with 1 to 4 GB of persistent RAM, which is affordable today. As of October 2001, we can add 2 GB of battery-backed RAM to our desktop computers and deploy *Conquest* for under \$200 [32]. Extending our design to other environments will be future work.

We will first present the design of *Conquest*, followed by a discussion of major design decisions.

### 3.1 File System Design

In our current design, *Conquest* stores all small files, metadata, executables, and shared libraries in persistent RAM; disks hold only the data content of remaining large files. We will discuss this media usage strategy further in Section 3.2.

An in-core file is stored logically contiguously in persistent RAM. Disks store only the data content of large files with coarse granularity, thereby reducing management overhead. For each large file, *Conquest* maintains a segment table in persistent RAM. On-disk allocation is done contiguously whenever possible in temporal order, similar to LFS [35] but without the need to perform continuous disk cleaning in the background.

For each directory, *Conquest* maintains a variant of an extensible hash table for its file metadata entries, with file names as keys. Hard links are supported by allowing multiple names (potentially under different directories) to hash to the same file metadata entry.

RAM storage allocation uses existing mechanisms in the memory manager when possible to avoid duplicate functionality. For example, the storage manager is relieved of maintaining a metadata

allocation table and a free list by using the memory address of the file metadata as its unique ID.

Although it reuses the code of the existing memory manager, *Conquest* has its own dedicated instances of the manager, residing persistently inside *Conquest*, each governing its own memory region. Paging and swapping are disabled for *Conquest* memory, but enabled for the non-*Conquest* memory region.

Unlike caching, RAM drives, and RAM file systems, *Conquest* memory is the final storage destination for many files and all metadata. We can access the critical path of *Conquest*'s main store without disk-related complexity in data duplication, migration, translation, synchronization, and associated management. Unlike RAM drives and RAM file systems, *Conquest* provides persistence and storage capacity beyond the size limitation of the physical main store.

### 3.2 Media-Usage Strategy

The first major design decision of *Conquest* is the choice of which data to place on disk, and the answer depends on the characteristics of popular workloads. Recent studies [9, 34, 42] independently confirm the often-repeated observations [30]:

- Most files are small.
- Most accesses are to small files.
- Most storage is consumed by large files, which are, most of the time, accessed sequentially.

Although one could imagine many complex data-placement algorithms (including LRU-style migration of unused files to the disk), we have taken advantage of the above characteristics by using a simple threshold to choose which files are candidates for disk storage. Only the data content of files above the threshold (currently 1 MB) are stored on disk. Smaller files, as well as metadata, executables, and libraries, are stored in RAM. The current choice of threshold works well, leaving 99% of all files in RAM in our tests. By enlarging this threshold, *Conquest* can incrementally use more RAM storage as the price of RAM declines. The current threshold was chosen somewhat arbitrarily, and future research will examine its appropriateness.

The decision to use a threshold simplifies the code, yet does not waste an unreasonable amount of memory since small files do not consume a large amount of total space. An additional advantage of the size-based threshold is that all on-disk files are large, which allows us to achieve significant simplifications in disk layout. For example, we can avoid adding complexity to handle fragmentation with "large" and "small" disk blocks, as in FFS [20]. Since we assume cheap and abundant RAM, the advantages of using a threshold far outweigh

the small amount of space lost by storing rarely used files in RAM.

### 3.2.1 Files Stored in Persistent RAM

Small files and metadata benefit the most from being stored in persistent RAM, given that they are more susceptible to disk-related overheads. Since persistent RAM access granularity is byte-oriented rather than block-oriented, a single-byte access can be six orders of magnitude faster than accessing disk [23].

Metadata no longer have dual representations, one in memory and one on disk. The removal of the disk representation also removes the complex synchronous or asynchronous mechanisms needed to propagate the metadata changes to disk [20, 12, 38], and avoids translation between the memory and disk representations.

At this time, *Conquest* does not give special treatment to executables and shared libraries by forcing them into memory, but we anticipate benefits from doing so. In-place execution will reduce startup costs and the time involved in faulting pages into memory during execution. Since shared libraries are modular extensions of executables, we intend to store them in-core as well.<sup>1</sup>

### 3.2.2 Large-File-Only Disk Storage

Historically, the handling of small files has been one major source of file system design complexity. Since small files are accessed frequently, and a small transfer size makes mechanical overheads significant, designers employ various techniques to speed up small-file accesses. For example, the content of small files can be stored in the metadata directly, or a directory structure can be mapped into a balanced tree on disk to ensure a minimum number of indirections before locating a small file [26]. Methods to reduce the seek time and rotational latency [20] are other attempts to speed up small-file accesses.

Small files introduce significant storage overhead because optimal disk-access granularities tend to be large and fixed, causing excessive internal fragmentation. Although reducing the access granularity necessitates higher overhead and lower disk bandwidth, the common remedy is nevertheless to introduce sub-granularities and extra management code to handle small files.

Large-file-only disk storage can avoid all these small-file-related complexities, and management overhead can be reduced with coarser access granularity. Sequential-access-mostly large files

exhibit well-defined read-ahead semantics. Large files are also read-mostly and incur little synchronization-related overhead. Combined with large data transfers and the lack of disk arm movements, disks can deliver near raw bandwidth when accessing such files.

## 3.3 Metadata Representation

How file system metadata is handled is critical, since this information is in the path of all file accesses. Below, we outline how *Conquest* optimizes behavior by its choices of metadata representation.

### 3.3.1 In-Core File Metadata

One major simplification of our metadata representation is the removal of nested indirect blocks from the commonly used *i*-node design. *Conquest* stores small files, metadata, executables, and shared libraries in persistent RAM, via uniform, single-level, dynamically allocated index blocks, so in-core data blocks are virtually contiguous.

*Conquest* does not use the *v*-node data structure provided by VFS to store metadata, because the *v*-node is designed to accommodate different file systems with a wide variety of attributes. Also, *Conquest* does not need many mechanisms involved in manipulating *v*-nodes, such as metadata caching. *Conquest*'s file metadata consists of only the fields (53 bytes) needed to conform to POSIX specifications.

To avoid file metadata management, we use the memory addresses of the *Conquest* file metadata as unique IDs. By leveraging the existing memory management code, this approach ensures unique file metadata IDs, no duplicate allocation, and fast retrieval of the file metadata. The downside of this decision is that we may need to modify the memory manager to anticipate that certain allocations will be relatively permanent.

For small in-core write requests where the total allocation is unknown in advance, *Conquest* allocates data blocks incrementally. The current implementation does not return unused memory in the last block of a file, though we plan to add automatic truncation as a future optimization. *Conquest* also supports "holes" within a file, since they are commonly seen during compilation and other activities.

### 3.3.2 Directory Metadata

We used a variant of extensible hashing [11] for our directory representation. The directory structure is built with a hierarchy of hash tables, using file names as keys. Collisions are resolved by splitting (or doubling) hash indices and unmasking an additional hash bit for each key. A path is resolved by recursively hashing

<sup>1</sup> Shared libraries can be trivially identified through magic numbers and existing naming and placement conventions.

each name component of the path at each level of the hash table.

Compared to `ext2`'s approach, hashing removes the need to compact directories that live in multiple (possibly indirect) blocks. Also, the use of hashing easily supports hard links by allowing multiple names to hash to the same file metadata entry. In addition, extendible hashing preserves the ordering of hashed items when changing the table size, and this property allows `readdir()` to walk through a directory correctly while resizing a hash table (e.g., recursive deletions).

One concern with using extensible hashing is the wasted indices due to collisions and subsequent splitting of hash indices. However, we found that alternative compact hashing schemes would consume similar amount of space to preserve ordering during a resize operation.

### 3.3.3 Large-File Metadata

For the data content of large files on disk, *Conquest* currently maintains a dynamically allocated, doubly linked list of segments to keep track of disk storage locations. Disk storage is allocated contiguously whenever possible, in temporal, or LFS, order [35].

Although we have a linear search structure, its simplicity and in-core speed outweigh its algorithmic inefficiency, as we will demonstrate in the performance evaluation (Section 5). In the worst case of severe disk fragmentation, we will encounter a linear slowdown in traversing the metadata. However, given that we have coarse disk-management granularity, the segment list is likely to be short. Also, since the search is in-core but access is limited by disk bandwidth, we expect little performance degradation for random accesses to large files.

Currently, we store the large-file data blocks sequentially as the write requests arrive, without regard to file membership. We chose this temporal order only for simplicity in the initial implementation. Unlike LFS, we keep metadata in-core, and existing file blocks are updated in-place as opposed to appending various versions of data blocks to the end of the log. Therefore, *Conquest* does not consume contiguous regions of disk space nearly as fast as LFS, and demands no continuous background disk cleaning.

Still, our eventual goal is to apply existing approaches from both video-on-demand (VoD) servers and traditional file systems research to design the final layout. For example, given its sequential-access nature, a large media file can be striped across disk zones, so disk scanning can serve concurrent accesses more effectively [8]. Frequently accessed large files can be stored completely near outer zones for higher disk bandwidth. Spatial and temporal ordering can be

applied within each disk zone, at the granularity of an enlarged disk block.

With a variety of options available, the presumption is that after enlarging the disk access granularity for large file accesses, disk transfer time will dominate access times. Since most large files are accessed sequentially, IO buffering and simple predictive prefetching methods should still be able to deliver good read bandwidth.

## 3.4 Memory Management

Although it reuses the code of the existing memory manager, *Conquest* has its own dedicated instances of the manager, residing completely in *Conquest*, with each governing its own memory region. Since all references within a *Conquest* memory manager are encapsulated within its governed region, and each region has its own dedicated physical address space, we can save and restore the runtime states of a *Conquest* memory manager directly in-core without serialization and deserialization.

*Conquest* avoids memory fragmentation by using existing mechanisms built in various layers of the memory managers under Linux. For sub-block allocations, the slab allocator compacts small memory requests according to object types and sizes [4]. For block-level allocations, memory mapping assures virtual contiguity without external fragmentation.

In the case of in-core storage depletion, we have several options. The simplest handling is to declare the resource depleted, which is our current approach (the same as is used for PDAs). However, under *Conquest*, this option implies that storage capacity is now limited by both memory and disk capacities. Dynamically adjusting the in-core storage threshold is another possibility, but changing the threshold can potentially lead to a massive migration of files. Our disk storage is potentially threatened with smaller-than-expected files and associated performance degradation.

## 3.5 Reliability

Storing data in-core inevitably raises the question of reliability and data integrity. At the conceptual level, disk storage is often less vulnerable to corruption by software failures because it is less likely to perform illegal operations through the rigid disk interface, unless memory-mapped. Main memory has a very simple interface, which allows a greater risk of corruption. A single wild kernel pointer could easily destroy many important files. However, a study conducted at the University of Michigan has shown that the risk of data corruption due to kernel failures is less than one might expect. Assuming one system crash



every two months, one can expect to lose in-memory data about once a decade [27].

Another threat to the reliability of an in-memory file system is the hardware itself. Modern disks have a mean time between failures (MTBF) of 1 million hours [37]. Two hardware components, the RAM and the battery backup system, cause *Conquest's* MTBF to be different from that of a disk. In our prototype, we use a UPS as the battery backup. The MTBF of a modern UPS is lower than that of disks, but is still around 100,000 hours [14, 36]. The MTBF of the RAM is comparable to disk [22]; however, the MTBF of *Conquest* is dominated by the characteristics of the complete computer system; modern machines again have an MTBF of over 100,000 hours. Thus, it can be seen that *Conquest* should lose data due to hardware failures at most once every few years. This is well within the range that users find acceptable in combination with standard backup procedures.

At the implementation level, an extension is to use approaches similar to Rio [7], which allows volatile memory to be used as a persistent store with little overhead. For metadata, we rely heavily on atomic pointer commits. In the event of crashes, the system integrity can remain intact, at the cost of potential memory leaks (which can be cleaned by fsck) for in-transit memory allocations.

In addition, we can still apply the conventional techniques of sandboxing, access control, checkpointing, fsck, and object-oriented self-verification. For example, *Conquest* still needs to perform system backups. *Conquest* uses common memory protection mechanisms by having a dedicated memory address space for storage (assuming a 64-bit address space). A periodic fsck is still necessary, but it can run at memory speed. We are also exploring the object-store approach of having a "typed" memory area, so a pointer can be verified to be of a certain type before being accessed.

### 3.6 64-Bit Addressing

Having a dedicated physical address space in which to run *Conquest* significantly reduces the 32-bit address space and raises the question of 64-bit addressing. However, our current implementation on a 32-bit machine demonstrates that 64-bit addressing implications are largely orthogonal to materializing *Conquest*, although a wide address space does offer many future extensions (i.e., having distributed *Conquest* sharing the same address space, so pointers can be stored directly and transferred across machine boundaries as in [6].)

## 4 *Conquest* Implementation Status

The *Conquest* prototype is operational as a loadable kernel module under Linux 2.4.2. The current implementation follows the VFS API, but we need to override generic file access routines at times to provide both in-core and on-disk accesses. For example, inside the read routine, we assume that accessing memory is the common case, while providing a forwarding path for disk accesses. The in-core data path no longer contains code for checking the status of the buffer cache, faulting and prefetching pages from disk, flushing dirty pages to disk to make space, performing garbage collection, and so on. The disk data path no longer contains mechanisms for on-disk metadata chasing and various internal fragmentation and seek-time optimizations for small files.

Because we found it relatively difficult to alter the VFS to not cache metadata, we needed to pass our metadata structures through VFS calls such as *mknod*, *unlink*, and *lookup*. We altered the VFS, so that the *Conquest* root node and metadata are not destroyed at *umount* times.

We modified the Linux memory manager in several ways. First, we introduced *Conquest* zones. With the flexibility built into the Linux zone allocator, it is feasible to allocate unused *Conquest* memory within a zone to perform other tasks such as IO buffering and program execution. However, we chose to manage memory at the coarser grain of zones, to conserve memory in a simpler way.

The *Conquest* memory manager is instantiated top-down instead of bottom-up, meaning *Conquest* uses high-level slab allocator constructs to instantiate dedicated *Conquest* slab managers, then lower-level zone and page managers. By using high-level constructs, we only need to build an instantiation routine, invoked at file system creation times.

Since *Conquest* managers reside completely in the memory region they govern, runtime states (i.e., pointers) of *Conquest* managers can survive reboots with only code written for reconnecting several data structure entry points back to *Conquest* runtime managers. No pointer translation was required.

*Conquest* is POSIX-compliant and supports both in-core and on-disk storage. We use a 1-MB static dividing line to separate small files from large files (Section 3.2). Large files are stored on disk in 4-KB blocks, so that we can use the existing paging and protection code without alterations. An optimization is to enlarge the block size to 64 KB or 256 KB for better performance.

## 5 Conquest Performance

We compared *Conquest* with *ext2* [5], *reiserfs* [26], *SGI XFS* [40], and *ramfs* by Transmeta. We chose *ext2*, *reiserfs*, and *SGI XFS* largely because they are the common basis for various performance comparisons. Note that with 2 Gbytes of physical RAM, these disk-based file systems use caching extensively, and our performance numbers reflect how well these file systems exploit memory hardware. In the experiments, all file systems have the same amount of memory available as *Conquest*.

*Ramfs* by Transmeta uses the page cache and v-nodes to store the file system content and metadata directly, and *ramfs* provides no means of achieving data persistence after a system reboot. Given that both *Conquest* and *ramfs* are under the VFS API and various OS legacy constraints, *ramfs* should approximate the practical achievable bound for *Conquest* performance. Our experimental platform is described in Table 5.1. Various file system settings are listed in Table 5.2.

Experimental platform	
Manufacturer model	Dell PowerEdge 4400
Processor	1 GHz 32-bit Xeon Pentium
Processor bus	133 MHz
Memory	4x512 MB, Micron MT18LSDT6472G, SYNCH, 133 MHz, CL3, ECC
L2 cache	256 KB Advanced
Disk	73.4 GB, 10,000 RPM, Seagate ST173404LC
Disk partition for testing	6.1 GB partition starting at cylinder 7197
I/O adaptor	Adaptec AIC-7899 Ultra 160/m SCSI host Adaptor, BIOS v25306
UPS	APC Smart-UPS 700
OS	Linux 2.4.2

Table 5.1: Experimental platform.

File system settings	
<i>cfs</i>	creation: default, mount: default
<i>ext2fs</i> (0.5b)	creation: default, mount: default
<i>transmeta ramfs</i>	creation: default, mount: default
<i>reiserfs</i> (3.6.25)	creation: default, mount: -o notail
<i>SGI XFS</i> (1.0)	creation: -l size=32768b mount: -o logbufs=8, logbsize=32768

Table 5.2: File system settings.

We used the Sprite LFS microbenchmarks [35]. As for macrobenchmarks, the most widely used in the file system literature is the Andrew File System Benchmark [16]. Unfortunately, this benchmark no longer stresses modern file systems because its data set is too small. Instead, we present results from the PostMark

macrobenchmark<sup>2</sup> [18] and our modified PostMark macrobenchmark, which is described in Section 5.3. All results are presented at a 90% confidence level.

### 5.1 Sprite LFS Microbenchmarks

The Sprite LFS microbenchmarks measure the latency and throughput of various file operations, and the benchmark suite consists of two separate tests for small and large files.

#### 5.1.1 Small-File Benchmark

The small-file benchmark measures the latency of file operations, and consists of creating, reading, and unlinking 10,000 1-KB files, in three separate phases. Figure 5.1 summarizes the results.

**Conquest vs. ramfs:** Compared to *ramfs*, *Conquest* incurs 5% and 13% overheads in file creation and deletion respectively, because *Conquest* maintains its own metadata and hashing data structures to support persistence, which is not provided by *ramfs*. Also, we have not removed or disabled VFS caching for metadata; therefore, VFS needs to go through an extra level of indirection to access *Conquest* metadata at times, while *ramfs* stores its metadata in cache.

Nevertheless, *Conquest* has demonstrated a 15% faster read transaction rate than *ramfs*, even when *ramfs* is performing at near-memory bandwidth. *Conquest* is able to improve this aspect of performance because the critical path to the in-core data contains no generic disk-related code, such as readahead and checking for cache status.

**Conquest vs. disk-based file systems:** Compared to *ext2*, *Conquest* demonstrates a 50% speed improvement for creation and deletion, mostly attributable to the lack of synchronous metadata manipulations. Like *ramfs*, *ext2* uses the generic disk access routines provided by VFS, and *Conquest* is 19% faster in read performance than cached *ext2*.

The performance of *SGI XFS* and *reiserfs* is slower than *ext2* because of both journaling overheads and their in-memory behaviors. *Reiserfs* actually achieved poorer performance with its original default settings. Interestingly, *reiserfs* performs better with the *notail* option, which disables certain disk optimizations for small files and the fractional block at the end of large files. While the intent of these disk optimizations is to save extra disk accesses, their overhead outweighs the benefits when there is sufficient memory to buffer disk accesses.

<sup>2</sup> As downloaded, Postmark v1.5 reported times only to a 1-second resolution. We have altered the benchmark to report timing data at the resolution of the system clock.

As for *SGI XFS*, its original default settings also produced poorer performance, since journaling consumes the log buffer quite rapidly. As we increased the buffer size for logging, *SGI XFS* performance improved. The numbers for both *reiserfs* and *SGI XFS* suggest that the overhead of journaling is very high.

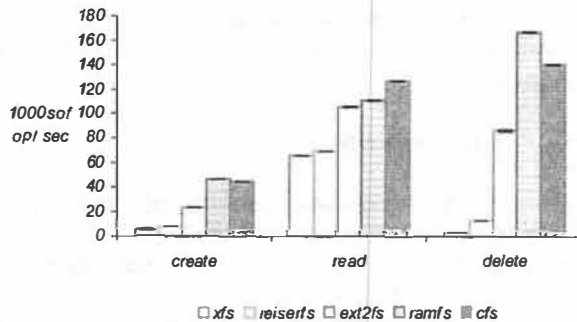


Figure 5.1: Transaction rate for the different phases of the Sprite LFS small-file benchmark, run over *SGI XFS*, *reiserfs*, *ext2*, *ramfs*, and *Conquest*. The benchmark creates, reads, and unlinks 10,000 1-KB files in separate phases. In this and most subsequent figures, the 90% confidence bars are nearly invisible due to the narrow confidence intervals.

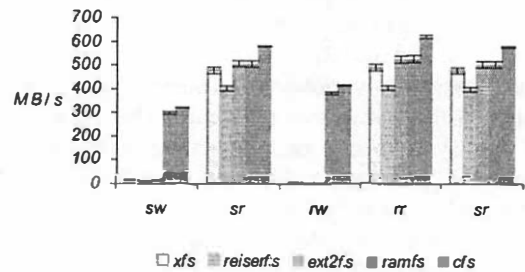
### 5.1.2 Large-File Benchmark

The large-file benchmark writes a large file sequentially (with flushing), reads from it sequentially, and then writes a new large file randomly (with flushing), reads it randomly, and finally reads it sequentially. The final read phase was originally designed to measure sequential read performance after random write requests were sequentially appended to the log in a log-structured file system. Data was flushed to disk at the end of each write phase.

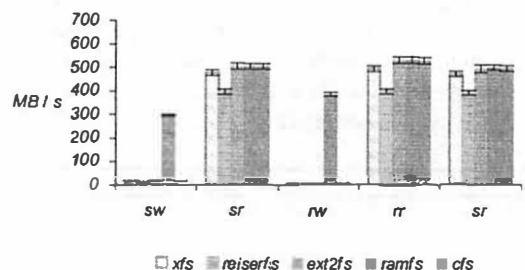
For *Conquest* on-disk files, we altered the large-file benchmark to perform each phase of the benchmark on forty 100-MB files before moving to the next phase. Since we have a dividing line between small and large files, we also investigated the sizes of 1 MB and 1.01 MB, with each phase of benchmark performed on ten 1-MB or 1.01-MB files. In addition, we memory-aligned all random accesses to reflect real-world usage patterns.

**The 1-MB benchmark:** The 1-MB large-file benchmark measures the throughput of *Conquest*'s in-core files (Figure 5.2a). Compared to *ramfs*, *Conquest* achieves 8% higher bandwidth in both random and sequential writes and 15% to 17% higher bandwidth in both random and sequential reads. It is interesting to observe that random memory writes and reads are faster than corresponding sequential accesses. This is because of cache hits: for 1-MB memory accesses with a 256-KB L2 cache size, random accesses have a roughly 25% chance of reusing the L2 cache content. We believe that the difference is larger for writes because of a write-back, write-allocate L2 cache design, which

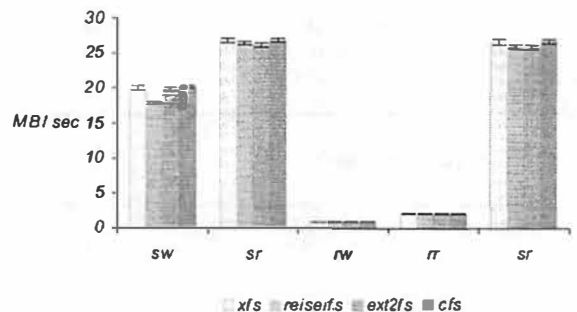
incurs additional overhead on sequential writes of large amounts of data.



(a) Sprite LFS large-file benchmark for 1-MB (in-core *Conquest*) files.



(b) Sprite LFS large-file benchmark for 1.01MB (on-disk *Conquest*) files.



(c) Sprite LFS large-file benchmark for 100-MB (on-disk *Conquest*) files.

Figure 5.2: Bandwidth for the different phases (sequential write, sequential read, random write, random read, sequential read) of the Sprite LFS large-file benchmarks, run over *SGI XFS*, *reiserfs*, *ext2*, *ramfs*, and *Conquest*. These two tests compare the performance of in-core and on-disk files under *Conquest*.

Compared to disk-based file systems, *Conquest* demonstrates a 1900% speed improvement in sequential writes over *ext2*, 15% in sequential reads, 6700% in random writes, and 18% in random reads. *SGI XFS* and *reiserfs* perform either comparably to or slower than *ext2*.



**The 1.01-MB benchmark:** The 1.01-MB large-file benchmark shows the performance effects of switching a file from memory to disk under *Conquest* (Figure 5.2b). *Conquest* disk performance matches the performance of cached *ext2* pretty well. In our design, in-core and on-disk data accesses use disjoint data paths wherever possible, so *Conquest* imposes little or no extra overhead for disk accesses.

**The 100-MB benchmark:** The 100-MB large-file benchmark measures the throughput of *Conquest* on-disk files (Figure 5.2c). We only compared against disk-based file systems because the total size exercised by the benchmark exceeds the capacity of *ramfs*. All file systems demonstrate similar performance. Compared to cached *ext2*, *Conquest* shows only 8% and 4% improvements in sequential and random writes. We expect further performance improvements after enlarging the block size to 64 KB or 256 KB.

## 5.2 PostMark Macrobenchmark

The PostMark benchmark was designed to model the workload seen by Internet service providers [18]. Specifically, the workload is meant to simulate a combination of electronic mail, netnews, and web-based commerce transactions.

PostMark creates a set of files with random sizes within a set range. The files are then subjected to transactions consisting of a pairing of file creation or deletion with file read or append. Each pair of transactions is chosen randomly and can be biased via parameter settings. The sizes of these files are chosen at random and are uniformly distributed over the file size range. A deletion operation removes a file from the active set. A read operation reads a randomly selected file in entirety. An append operation opens a random file, seeks to the end of the file, and writes a random amount of data, not exceeding the maximum file size.

We initially ran our experiments using the configuration of 10,000 files with a size range of 512 bytes to 16 KB. One run of this configuration performs 200,000 transactions with equal probability of creates and deletes, and a four times higher probability of performing reads than appends. The transaction block size is 512 bytes. However, since this workload is far smaller than the workload observed at any ISP today, we varied the total number of files from 5,000 to 30,000 to see the effects of scaling.

Another adjustment of the default setting is the assumption of a single flat directory. Since it is unusual to store 5,000 to 30,000 files in a single directory, we reconfigured PostMark to use one subdirectory level to distribute files uniformly, with the number of directories equal to the square root of the file set size.

This setting ensures that each level has the same directory fanout.

Since all files within the specified size range will be stored in memory under *Conquest*, this benchmark does not exercise the disk aspect of the *Conquest* file system. Also, since this configuration specifies an average file set of only 250 MB, which fits comfortably in 2 GB of memory, this benchmark compares the memory performance of *Conquest* against the performance of existing cache and IO buffering mechanisms, under a realistic mix of file operations.

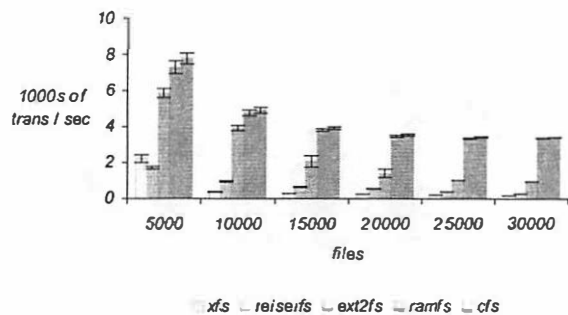


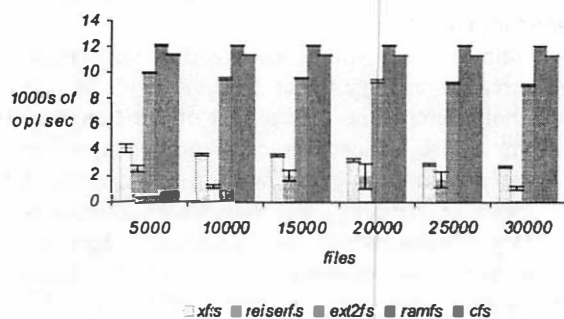
Figure 5.3: PostMark transaction rate for SGI XFS, *reiserfs*, *ext2*, *ramfs*, and *Conquest*, varying from 5,000 and 30,000 files. The results are averaged over five runs.

Figure 5.3 compares the transaction rates of *Conquest* with various file systems as the number of files is varied from 5,000 to 30,000. First, the performance of *Conquest* differs little from *ramfs* performance. We feel comfortable with *Conquest*'s performance at this point, given that we still have room to reduce costs for at least sequential writes (enlarging the disk block size). *Conquest* outperforms *ext2* significantly; the performance gap widens from 24% to 350% as the number of files increases. SGI XFS and *reiserfs* perform slower than *ext2* due to journaling overheads.

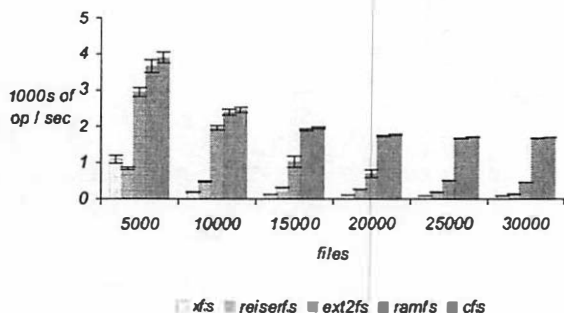
For space reasons, we have omitted other graphs with similar trends—bandwidth, average creation rate, read rate, append rate, and average deletion rate.

Taking a closer look at the file-creation component of the performance numbers, we can see that without interference from other types of file transactions (Figure 5.4a), file creation rates show little degradation for all systems as the number of files increases. When mixed with other types of file transactions (Figure 5.4b), file creation rates degrade drastically.

With only file creations, *Conquest* creates 9% fewer files per second than *ramfs*. However, when creations are mixed with other types of file transactions, *Conquest* creates files at a rate comparable to *ramfs*.



(a) PostMark file creation rate.



(b) PostMark file creation rate, mixed with other types of file transactions.

Figure 5.4: PostMark file creation performance for *SGI XFS*, *reiserfs*, *ext2*, *ramfs*, and *Conquest*, varying from 5,000 and 30,000 files. The results are averaged over five runs.

Compared to *ext2*, *Conquest* performs at a 26% faster creation rate (Figure 5.4a), compared to the 50% faster rate in the LFS Sprite benchmark. *Ext2* has a better creation rate under PostMark because files being created have larger file sizes. The write buffer used by *ext2* narrows the performance difference of file creation when compared to *Conquest*.

Similar to the comparison between *Conquest* and *ramfs*, it is interesting to see that *SGI XFS* has a faster file creation rate than *reiserfs* without mixed traffic, but a slower rate than *reiserfs* with mixed traffic. This result demonstrates that optimizing individual operations in isolation does not necessarily produce better performance when mixed with other operations.

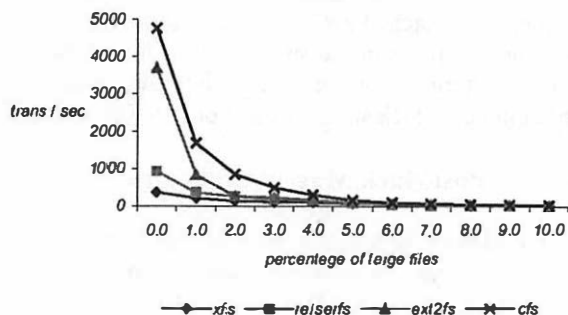
We have omitted the graphs for file deletion, since they show similar trends.

### 5.3 Modified Postmark Benchmark

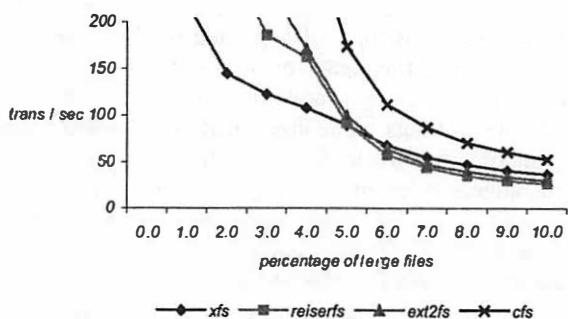
To exercise both the memory and disk components of *Conquest*, we modified the Postmark benchmark in the following way. We generated a percentage of files in a large-file category, with file sizes uniformly distributed between 2 MB and 5MB. The remaining files were uniformly distributed between 512 bytes to 16 KB. We

fixed the total number of files at 10,000 and varied the percentage of large files from 0.0 to 10.0 (0 GB to 3.5 GB). Since the file set exceeds the storage capacity of *ramfs*, we were forced to omit *ramfs* from our results.

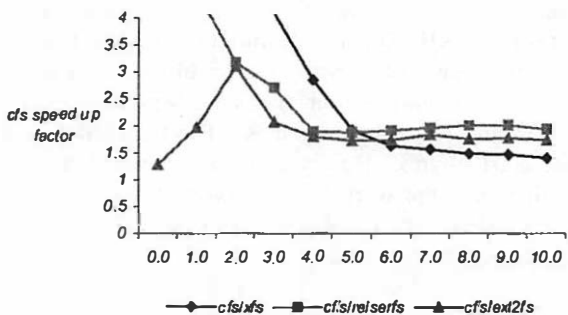
Figure 5.6 compares the transaction rate of *SGI XFS*, *reiserfs*, *ext2*, and *Conquest*. Figure 5.6a shows how the measured transaction rates of the four file systems vary as the percentage of large files increases. Because the scale of this graph obscures important detail at the right-hand side, Figure 5.6b zooms into the graph with an expanded vertical scale. Finally, Figure 5.6c shows the performance ratio of *Conquest* over other disk-based file systems.



(a) The full-scale graph.



(b) The zoomed graph.



(c) *Conquest* speedup curves for the full graph.

Figure 5.6: Modified PostMark transaction rate for *SGI XFS*, *reiserfs*, *ext2*, and *Conquest*, with varying percentages of large (on-disk *Conquest*) files ranging from 0.0 to 10.0 percent.

*Conquest* demonstrates 29% to 310% faster transfer rates than *ext2* (Figure 5.6c). The shape of the *Conquest* speedup curve over *ext2* reflects the rapid degradation of *ext2* performance with the injection of disk traffic. As more disk traffic is injected, we start to see a relatively steady performance ratio. At steady state, *Conquest* shows a 75% faster transaction rate than *ext2*.

Both *SGI XFS* and *reiserfs* show significantly slower memory performance (left side of Figure 5.6a). However, as the file set exceeds the memory size, *SGI XFS* starts to outperform *ext2* and *reiserfs* (Figure 5.6c). Clearly, different file systems are optimized for different conditions.

## 6 Related Work

The database community has a long established history of memory-only systems. An early survey paper reveals key architectural implications of sufficient RAM and identifies several early main memory databases [13]. The cost of main memory may be the primary concern that prevents operating systems from adopting similar solutions for practical use, and *Conquest* offers a transition for delivering file system services from main memory in a practical and cost-effective way.

In the operating system arena, one early use of persistent RAM was for buffering write requests [2]. Since dirty data were buffered in persistent memory, the interval between synchronizations to the disk could be lengthened.

The Rio file cache [28] combines UPS, volatile memory, and a modified write-back scheme to achieve the reliability of write-through file cache and performance of pure write-back file cache (with no reliability-induced writes to disk). The resiliency offered by Rio complements *Conquest*'s performance well. While *Conquest* uses main store as the final storage destination, Rio's BIOS *safe sync* mechanism provides a high assurance of dumping *Conquest* memory to disk in the event of infrequent failures that require power cycles.

Persistent RAM has been gaining acceptance as the primary storage medium on small mobile computing devices through a plethora of flash-memory-based file systems [29, 41]. Although this departure from disk storage marks a major milestone toward persistent-RAM-based storage, flash memory has some unpleasant characteristics, notably the limited number of erase-write cycles and slow (second-range) time for storage reclamation. These characteristics cause performance problems and introduce a different kind of operating system complexity. Our research currently

focuses on the general performance characteristics exemplified by battery-backed DRAM (BB-DRAM).

The leading PDA operating systems, PalmOS and Windows CE, deliver memory and file system services via BB-DRAM, but both systems are more concerned with fitting an operating system into a memory-constrained environment, in contrast to the assumed abundance of persistent RAM under *Conquest*. PalmOS lacks a full-featured execution model, and efficient methods for accessing large data objects are limited [1]. Windows CE is unsuitable for general desktop-scale deployment because it tries to shrink the full operating system environment to the scale of a PDA. Also, the Windows CE architecture inherits many disk-related assumptions [24].

IBM AS/400 servers provide the appearance of storing all files in memory from the user's point of view. This uniform view of storage access is accomplished by the extensive use of virtual memory. The AS/400 design is an example of how *Conquest* can enable a different file system API. However, underneath the hood of AS/400, conventional roles of memory acting as the cache for disk content still apply, and disks are still the persistent storage medium for files [17].

One form of persistent RAM under development is Magnetic RAM (MRAM) [3]. An ongoing project on MRAM-enabled storage, HeRMES, also takes advantage of persistent RAM technologies [25]. HeRMES uses MRAM primarily to store the file metadata to reduce a large component of existing disk traffic, and also to buffer writes to lengthen the time frame for committing modified data. HeRMES also assumes that persistent RAM will remain a relatively scarce resource for the foreseeable future, especially for large file systems.

## 7 Lessons Learned

Through the design and implementation of *Conquest*, we have learned the following major lessons:

First, the handling of disk characteristics permeates file system design even at levels above the device layer. For example, default VFS routines contain readahead and buffer-cache mechanisms, which add high and unnecessary overheads to low-latency main store. Because we needed to bypass these mechanisms, building *Conquest* was much more difficult than we initially expected. For example, certain downstream storage routines anticipate data structures associated with disk handling. We either need to find ways to reuse these routines with memory data structures, or construct memory-specific access routines from scratch.

Second, file systems that are optimized for disk are not suitable for an environment where memory is

abundant. For example, *reiserfs* and *SGI XFS* do not exploit the speed of RAM as well as we anticipated. Disk-related optimizations impose high overheads for in-memory accesses.

Third, matching the physical characteristics of media to storage objects provides opportunities for faster performance and considerable simplification for each medium-specific data path. *Conquest* applies this principle of specialization: leaving only the data content of large files on disk leads to simpler and cleaner management for both memory and disk storage. This observation may seem obvious, but results are not automatic. For example, if the cache footprint of two specialized data paths exceeds the size of a single generic data path, the resulting performance can go in either direction, depending on the size of the physical cache.

Fourth, access to cached data in traditional file systems incurs performance costs due to commingled disk-related code. Removing disk-related complexity for in-core storage under *Conquest* therefore yields unexpected benefits even for cache accesses. In particular, we were surprised to see *Conquest* outperform *ramfs* by 15% in read bandwidth, knowing that storage data paths are already heavily optimized.

Finally, it is much more difficult to use RAM to improve disk performance than it might appear at first. Simple approaches such as increasing the buffer cache size or installing simple RAM-disk drivers do not generate a full-featured, high-performance solution.

The overall lesson that can be drawn is that seemingly simple changes can have much more far-reaching effects than first anticipated. The modifications may be more difficult than expected, but the benefits can also be far greater.

## 8 Future Work

*Conquest* is now operational, but we can further improve its performance and usability in a number of ways. A few previously mentioned areas are designing mechanisms for adjusting file size threshold dynamically (Section 3.4) and finding a better disk layout for large data blocks (Section 3.3.3).

High-speed in-core storage also opens up additional possibilities for operating systems. *Conquest* provides a simple and efficient way for kernel-level code to access a general storage service, which is conventionally either avoided entirely or achieved through the use of more limited buffering mechanisms. One major area of application for this capability would be system monitoring and lightweight logging, but there are numerous other possibilities.

In terms of research, so far we have aggressively removed many disk-related complexities from the in-

core critical path without questioning exactly how much each disk optimization adversely affects file system performance. One area of research is to break down these performance costs, so designers can improve the memory performance for disk-based file systems.

Memory under *Conquest* is a shared resource among execution, storage, and buffering for disk access. Finding the “sweet spot” for optimal system performance will require both modeling and empirical investigation. In addition, after reducing the roles of disk storage, *Conquest* exhibits different system-wide performance characteristics, and the implications can be subtle. For example, the conventional wisdom of mixing CPU- and IO-bound jobs may no longer be a suitable scheduling policy. We are currently experimenting with a wider variation of workloads to investigate a fuller range of *Conquest* behavior.

## 9 Conclusion

We have presented *Conquest*, a fully operational file system that integrates persistent RAM with disk storage to provide significantly improved performance compared to other approaches such as RAM disks or enlarged buffer caches. With the involvement of both memory and disk components, we measure a 43% to 96% speedup compared to popular disk-based file systems.

During the development of *Conquest*, we discovered a number of unexpected results. Obvious ad hoc approaches not only fail to provide a complete solution, but perform more poorly than *Conquest* due to the unexpectedly high cost of going through the buffer cache and disk-specific code. We found that it was very difficult to remove the disk-based assumptions integrated into operating systems, a task that was necessary to allow *Conquest* to achieve its goals.

The benefits of *Conquest* arose from rethinking basic file system design assumptions. This success suggests that the radical changes in hardware, applications, and user expectations of the past decade should also lead us to rethink other aspects of operating system design.

## 10 Acknowledgements

We would like to thank our shepherd Darrell Anderson and the anonymous reviewers who have offered invaluable suggestions to strengthen this paper. We would also like to thank Michael Gorlick and Richard Guy for reviewing an early presentation of the *Conquest* performance results and offering useful

insights. In addition, we want to thank Mark Yarvis, Scott Michel, and Janice Wheeler for commenting on earlier drafts of this paper. This work was supported by the National Science Foundation under Grant No. CCR-0098363.

## 11 References

- [1] 3COM. Palm OS® Programmer's Companion. <http://www.palmos.com> (under site map and documentation), 2002.
- [2] Baker M, Asami S, Deprit E, Ousterhout J, Seltzer M. Non-Volatile Memory for Fast, Reliable File Systems. *Proceedings of the 5<sup>th</sup> International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.
- [3] Boeve H, Bruynseraede C, Das J, Dessein K, Borghs G, De Boeck J, Sousa R, Melo L, Freitas P. Technology assessment for the implementation of magnetoresistive elements with semiconductor components in magnetic random access memory (MRAM) architectures. *IEEE Transactions on Magnetics* 35(5), pp. 2820-2825, 1999.
- [4] Bonwick J. The Slab Allocator: An Object-Caching Kernel Memory Allocator. *Proceedings of USENIX Summer 1994 Technical Conference*, June 1994.
- [5] Card R, Ts'o T, Tweedie S. Design and Implementation of the Second Extended Filesystem. *The HyperNews Linux KHG Discussion*. <http://www.linuxdoc.org> (search for ext2 Card Tweedie design), 1999.
- [6] Chase J, Levy H, Lazowska E, Baker-Harvey M. Opal: A Single Address Space System for 64-Bit Architectures. *Proceedings of IEEE Workshop on Workstation Operating Systems*, April 1992.
- [7] Chen PM, Ng WT, Chandra S, Ayccock C, Rajamani G, Lowell D. The Rio File Cache: Surviving Operating System Crashes. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.
- [8] Chen S, Thapar M. A Novel Video Layout Strategy for Near-Video-on-Demand Servers. *Technical Report HPL-97-52*, 1997.
- [9] Douceur JR, Bolosky WJ. A Large-Scale Study of File-System Contents. *Proceedings of the ACM Sigmetrics '99 International Conference on Measurement and Modeling of Computer Systems*, May 1999.
- [10] Douglass F, Caceres R, Kaashoek F, Li K, Marsh B, Tauber JA. Storage Alternatives for Mobile Computers. *Proceedings of the 1<sup>st</sup> Symposium on Operating Systems Design and Implementation*, November 1994.
- [11] Fagin R, Nievergelt J, Pippenger N, Raymond Strong H. Extensible hashing—a fast access method for dynamic files, *ACM Transactions on Database Systems*, 4(3) pp. 315-344, 1979.
- [12] Ganger GR, McKusick MK, Soules CAN, Patt YN. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems*, 18(2) pp. 127-153, May 2000.
- [13] Garcia-Molina H, Salem K. *IEEE Transactions on Knowledge and Data Engineering*, 4(6) pp. 509-516, December 1992.
- [14] Gibson GA, Patterson DA. Designing Disk Arrays for High Data Reliability. *Journal of Parallel and Distributed Computing*, 1993.
- [15] Griffioen J, Appleton R. Performance Measurements of Automatic Prefetching. *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, September 1995.
- [16] Howard J, Kazar M, Menees S, Nichols D, Satyanarayanan M, Sidebotham R, West M. Scale and Performance in a Distributed File System, *ACM Transactions on Computer Systems*, 6(1), pp. 51-81, February 1988.
- [17] IBM® Server iSeries Storage Solutions. <http://www-1.ibm.com/servers/eserver/iseries/hardware/storage/overview.html>, 2002.
- [18] Katcher J. PostMark: A New File System Benchmark. *Technical Report TR3022*. Network Appliance Inc., October 1997.
- [19] Kroeger KM, Long DDE. Predicting File System Actions from Prior Events. *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.
- [20] McKusick MK, Joy WN, Leffler SJ, Fabry RS. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3), pp. 181-197, 1984.
- [21] McKusick MK, Karels MJ, Bostic K. A Pageable Memory Based Filesystem. *Proceeding of USENIX Conference*, June 1990.
- [22] Module Mean Time Between Failures (MTBF). Technical Note TN-04-45. <http://download.micron.com/pdf/technotes/DT45.pdf> (go to micron.com, and search for MTBF), 1997.
- [23] Micron DRAM Product Information. <http://www.micron.com> (under DRAM and data sheets), 2002.
- [24] Microsoft. MSDN Online Library, <http://msdn.microsoft.com/library> (under embedded development and Windows CE), 2002.

- [25] Miller EL, Brandt SA, Long DDE. HerMES: High-Performance Reliable MRAM-Enabled Storage. *Proceedings of the 8<sup>th</sup> IEEE Workshop on Hot Topics in Operating Systems*, May 2001.
- [26] Namesys. <http://www.namesys.com>, 2002.
- [27] Ng WT, Aycock CM, Rajamani G, Chen PM. Comparing Disk and Memory's Resistance to Operating System Crashes. *Proceedings of the 1996 International Symposium on Software Reliability Engineering*, October 1996.
- [28] Ng WT, Chen PM. The Design and Verification of the Rio File Cache. *IEEE Transactions on Computers*, 50(4), April 2001.
- [29] Nijima H. Design of a Solid-State File Using Flash EEPROM. *IBM Journal of Research and Development*. 39(5), September 1995.
- [30] Ousterhout JK, Da Costa H, Harrison D, Kunze A, Kupfer M, Thompson JG. A Trace Driven Analysis of the UNIX 4.2 BSD File Systems. *Proceedings of the 10<sup>th</sup> ACM Symposium on Operating Systems Principles*, pp. 15-24, December 1985.
- [31] Patterson RH, Gibson GA, Ginting E, Stodolsky D, Zelenka J. Informed Prefetching and Caching. *Proceedings of the 15<sup>th</sup> ACM Symposium on Operating Systems Principles*, pp. 79-95, December 1995.
- [32] Price Watch. New Computer Components. <http://www.pricewatch.com>, 2001.
- [33] Riedel E. A Performance Study of Sequential I/O on Windows NT 4. *Proceedings of the 2<sup>nd</sup> USENIX Windows NT Symposium*, Seattle, August 1998.
- [34] Roselli D, Lorch JR, Anderson TE. A Comparison of File System Workloads. *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.
- [35] Rosenblum M, Ousterhout J. The Design and Implementation of a Log-Structured File System. *Proceedings of the 13<sup>th</sup> ACM Symposium on Operating Systems Principles*, October 1991.
- [36] Power Conversion Systems, <http://www.schaeferpower.com/sminvter.htm> (Google keywords: schaeferpower, UPS, MIL-HDBK-217), 2000.
- [37] Barracuda Technical Specifications. <http://www.seagate.com> (click on find, barracuda, and technical specifications), 2002.
- [38] Seltzer MI, Ganger GR, McKusick MK, Smith KA, Soules CAN, Stein CA. Journaling Versus Soft Updates: Asynchronous Meta-Data Protection in File Systems. *Proceedings of 2000 USENIX Annual Technical Conference*, June 2000.
- [39] Steere DC. Exploiting the Non-Determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency. *Proceedings of the 16<sup>th</sup> ACM Symposium on Operating Systems Principles*, December 1997.
- [40] Sweeney A, Doucette D, Hu W, Anderson C, Nishimoto M, Peck G. Scalability in the XFS File System. *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.
- [41] Torelli P. The Microsoft Flash File System. *Dr. Dobbs's Journal*, pp. 63-70, February 1995.
- [42] Vogels W. File System Usage in Windows NT 4.0. *Proceedings of the 17<sup>th</sup> Symposium on Operating Systems Principles*, December 1999.
- [43] Wu M, Zwaenepoel W, eNvy: A Non-Volatile, Main Memory Storage System. *Proceedings of the 6<sup>th</sup> Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.
- [44] ZDNet. Quantum Rushmore Solid-State Disk. <http://www.zdnet.com/sp/stories/issue/0,4537,3963,22,00.html> (Google keywords: zdnet solid state disk), 1999.



# Exploiting Gray-Box Knowledge of Buffer-Cache Management

Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau  
*Department of Computer Sciences, University of Wisconsin–Madison*  
{ncb, johnbent, dusseau, remzi}@cs.wisc.edu

## Abstract

*The buffer-cache replacement policy of the OS can have a significant impact on the performance of I/O-intensive applications. In this paper, we introduce a simple fingerprinting tool, Dust, which uncovers the replacement policy of the OS. Specifically, we are able to identify how initial access order, recency of access, frequency of access, and long-term history are used to determine which blocks are replaced from the buffer cache. We show that our fingerprinting tool can identify popular replacement policies described in the literature (e.g., FIFO, LRU, LFU, Clock, Random, Segmented FIFO, 2Q, and LRU-K) as well as those found in current systems (e.g., NetBSD, Linux, and Solaris).*

*We demonstrate the usefulness of fingerprinting the cache replacement policy by modifying a web server to use this knowledge; specifically, the web server infers the contents of the OS file cache by modeling the replacement policy under the given set of page requests. We show that by first servicing those web pages that are believed to be resident in the OS buffer cache, we can improve both average response time and throughput.*

## 1 Introduction

Although the specific algorithms used to manage the buffer cache can significantly impact the performance of I/O-intensive applications [8, 13, 27], this knowledge is usually hidden from user processes. Currently, to determine the behavior of the buffer cache, implementors are forced to rely on available documentation, access to source code, or general knowledge of how buffer caches behave.

Rather than relying on these *ad hoc* methods, we propose the use of *fingerprinting* to automatically uncover characteristics of the OS buffer cache. In this paper, we describe *Dust*, a simple fingerprinting tool that is able to identify the buffer-cache replacement policy; specifically, we identify whether it uses initial access order, recency of access, frequency of access, or historical information.

Fingerprinting can be described as the use of micro-benchmarking techniques to identify the algorithms and policies used by the system under test. The idea behind

fingerprinting is to insert *probes* into the underlying system and to observe the resulting behavior through visible outputs. By carefully controlling the probes and matching the resulting output to the fingerprints of known algorithms, one can often identify the algorithm of the system under test. The key challenge is to inject probes to create distinctive fingerprints such that different algorithmic characteristics can be isolated.

There are several significant advantages to using fingerprints for automatically identifying internal algorithms. First, fingerprinting eliminates the need for a developer to obtain documentation or source code to understand the underlying system. Second, fingerprinting enables all programmers, not just those with sophisticated experience, to use algorithmic knowledge and thus improve performance. Third, fingerprinting can uncover bugs, or hidden complexities, in systems either under development or already deployed. Finally, fingerprinting can be used at run-time, allowing an adaptive application to modify its own behavior based on the characteristics of the underlying system.

In this paper, we investigate a new use of algorithmic knowledge: its use in exposing the current contents of the OS buffer cache. Recent work has shown that I/O-intensive applications can improve their performance given information about the contents of the file cache [3, 33]; specifically, applications that can handle data from disk in a flexible order should first access those blocks in the buffer cache and then those on disk. However, current approaches suffer from one of two limitations: they either require changes to the underlying OS to export this information or cannot accurately identify the presence of small files in the buffer cache.

We observe that an application can model (or simulate) the state of the buffer cache if it knows the replacement policy used by the OS and can see most file accesses. A dedicated web server can greatly benefit from knowing the contents of the buffer cache and servicing first those requests that will hit in the buffer cache. We have implemented a cache-aware web server based on the NeST storage appliance [6] and show that this web server improves both average response time and throughput.

In this paper we make the following contributions:

- We introduce *Dust*, a fingerprinting tool that automatically identifies cache replacement policies based upon how they prioritize between initial access order, recency of access, frequency of access, and historical information.
- We demonstrate through simulations that *Dust* can distinguish between a variety of replacement policies found in the literature: FIFO, LRU, LFU, Random, Clock, Segmented FIFO, 2Q, and LRU-K.
- We use our fingerprinting software to identify the replacement policies used in several operating systems: NetBSD 1.5, Linux 2.2.19 and 2.4.14, and Solaris 2.7.
- We show that by knowing the OS replacement policy, a cache-aware web server can first service those requests that can be satisfied within the OS buffer cache and thereby obtain substantial performance improvements.

The rest of this paper is organized as follows. We begin in Section 2 by describing our fingerprinting approach. In Section 3 we show via simulation that we can identify a range of popular replacement policies. In Section 4 we identify the replacement policies used in several current operating systems. In Section 5 we show how a web server can exploit knowledge of the buffer-cache replacement policy for improved performance. We briefly discuss related work in Section 6, and conclude in Section 7.

## 2 Fingerprinting Methodology

We now describe *Dust*, our software for identifying the page replacement policy employed by an operating system. By manipulating how blocks are accessed, forcing evictions, and then observing which blocks are replaced, *Dust* can identify the parameters used by the page replacement policy and the corresponding algorithm.

*Dust* relies upon probes to infer the current state of the buffer cache. By measuring the time to read a byte within a file block, one can determine whether or not that block was previously in the buffer cache. Intuitively, if the probe is “slow”, one infers that the block was previously on disk; if the probe is “fast”, then one infers that the block was already in the cache.

For *Dust* to correctly distinguish between different replacement policies, we must first identify the file block attributes used by existing policies to select a victim block for replacement. From a search of the OS and database research literature and the documentation of existing operating systems, we have identified four attributes that are often used for replacement: the order

of initial access to the block (*e.g.*, FIFO), the recency of accesses (*e.g.*, LRU), the frequency of accesses (*e.g.*, LFU) and historical accesses to blocks (*e.g.*, 2Q [12]). Thus, we can correctly identify the use of combinations of these four attributes within a replacement policy.

We note that some operating systems use replacement policies that consider attributes beyond what *Dust* considers. For example, some replacement policies consider whether or not pages are dirty [16], the size of the file the page is from, or replacement cost [10]. Further, replacement of pages can be performed on either a global or per process basis [14]. Finally, in real systems, not only are file pages cached, but file meta-data as well, and some systems prefer to evict pages from files whose meta-data is no longer cached. It is also possible that future replacement policies may utilize new attributes that we do not currently fingerprint. Although *Dust* can not currently identify these parameters, we believe that the basic framework within *Dust* can be extended to do so.

Given our goal of identifying replacement policies, there are three primary components to *Dust*. First, the size of the buffer cache is measured with a simple microbenchmark; this value is used as input to the remaining steps. Second, the short-term replacement algorithm is fingerprinted, based upon initial access, recency of access, and frequency of access. Third, *Dust* determines whether or not long-term history is used by the replacement algorithm.

### 2.1 Microbenchmarking Buffer Cache Size

To manipulate the state of the buffer cache and interpret its contents, *Dust* must first know the *size* of the buffer cache. Since this information is not readily available through a common interface on most systems, *Dust* contains a simple microbenchmark. *Dust* accesses progressively larger amounts of file data until it notices that some blocks no longer fit the cache. For each increase in the tested size, there are two steps. In the first step, *Dust* touches the file blocks up through the newly increased size to fetch them into the buffer cache. In the second step, *Dust* probes each block again, measuring the time per probe to verify if the block is still in the cache. This technique is similar to the technique used to determine available memory in NOW-Sort [4].

There are two important features of this approach. First, by probing *every* file block in the second step, this algorithm is independent of the replacement policy used to manage the buffer cache. Second, this algorithm works even when the buffer cache is integrated with the virtual memory system, assuming that *Dust* uses little memory and the buffer cache is able to grow to its maximum size. Further, as we will show, our fingerprinting algorithm is robust to slight inaccuracies in our estimation of the buffer cache size.



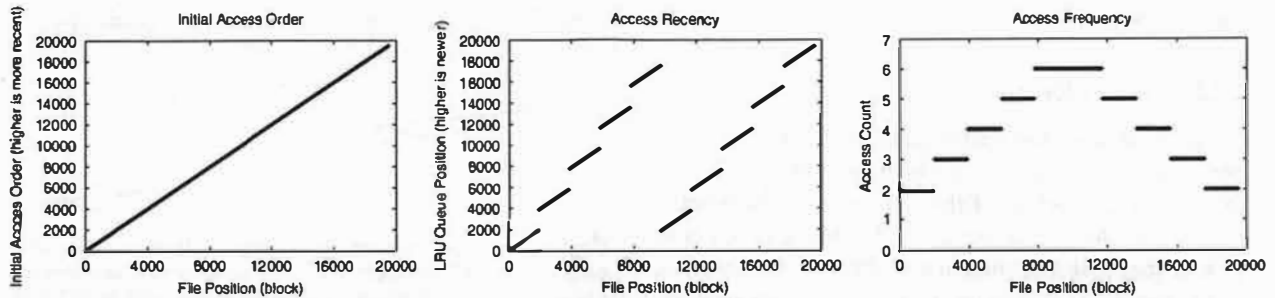


Figure 1: Short-Term Attributes of Blocks. The three graphs show the priority of each block within the test region according to the three metrics: order of initial access, recency of access, and frequency of access. The x-axis indicates the block number within the file forming the test region. The y-axis indicates the initial accesses order (left), recency of access (center) and frequency of access (right).

## 2.2 Fingerprinting Replacement Attributes

Once the buffer cache size is known, *Dust* determines the attributes of file blocks that are used by the OS short-term replacement policy. This fingerprinting stage involves three simple steps. First, *Dust* reads file blocks into the buffer cache while simultaneously controlling the replacement attributes of each block (e.g., by accessing blocks in different initial access, recency, and frequency orders). Second, *Dust* forces some of these blocks to be evicted from the buffer cache by accessing additional file data. Finally, the contents of the buffer cache are inferred by probing random sets of blocks; the cache state of these file blocks is then plotted to illustrate the replacement policy. We now describe each of these three steps in detail.

### 2.2.1 Configuring Attributes

The first step moves the buffer cache into a known and well-controlled state – both the data blocks that are resident and the initial access, recency, and frequency attributes of each resident block. This control is imposed by performing a pattern of reads over blocks within a single file; we refer to these blocks as the *test region*. To ensure that all of this data is resident, the size of this test region is set slightly smaller than the estimate of the buffer cache size (precisely, we use only 90% of the estimated cache size and adjust the size such that each of ten stripes discussed below are page aligned).

Controlling the initial access parameter of each block allows *Dust* to identify replacement policies that are based on the initial access order of blocks (e.g., FIFO). To exert this control, our access pattern begins with a sequential scan of the test region. The resulting initial access queue ordering is shown in the first graph of Figure 1; specifically, the blocks at the end of the file are those that are given priority (i.e., remain in the buffer cache) given a FIFO-based policy.

*Dust* is able to identify replacement policies that are

based on temporal locality (e.g., LRU) by controlling how recently each block is accessed and ensuring that this ordering does not match the initial access ordering. To ensure this criteria, a pattern of reads across ten stripes within the file are performed. Specifically, two indices into the file are maintained: a left pointer, which starts at the beginning of the file, and a right pointer, which starts at the center of the test region. The workload alternates between reading one stripe as indicated by the left pointer and then one stripe as indicated by the right pointer. The pattern continues until the left pointer reaches the center of the test region and the right pointer reaches the end. This controlled pattern of access induces the recency queue order shown in the middle graph of Figure 1; specifically, the blocks at the end of the left and right regions are those given priority with an LRU-based policy.

Finally, to identify policies that have a frequency based component, *Dust* ensures that stripes in the test region have distinctive frequency counts. When reading stripes for recency ordering, *Dust* touches each stripe multiple times for a frequency ordering as well. In our pattern, stripes near the center of the test region are read the most often, and those near the beginning and end of the test region are read the least. The number of reads for each area of the test region is shown in the right-most graph of Figure 1, where blocks in the middle are given priority with an LFU-based policy.

The need to impose different frequencies on different parts of the file is part of the motivation for dividing the test region into a fixed number of stripes. If, for instance, each block of the test region were given a different frequency count, the runtime of *Dust* would be exponential in the size of the file. In our simulation experiments, we determined ten to be a good number. The more stripes used, the more precise the fingerprint becomes since there is a greater variety of frequency and recency regimes. However, a greater number of stripes makes each stripe smaller thus making the data more

susceptible to noise.

### 2.2.2 Forcing Evictions

Once the state of the buffer cache is configured, *Dust* performs an *eviction scan* in which more file data is read to cause some portion of the test region to be evicted from the cache. Since the goal of evicting pages is to give us the most information and ability to differentiate across replacement policies, *Dust* tries to evict approximately half of the cached data.<sup>1</sup>

We note that the eviction scan must read each page multiple times such that the frequency counts of its pages are higher than those of the pages in the test region. Otherwise, *Dust* is not able to identify a frequency-based replacement policies since the eviction region would replace its own pages. This illustrates one of the limitations of our approach: we do not differentiate between LIFO, MRU, and MFU replacement policies, since all replace the eviction region with itself. However, we feel that this limitation is acceptable, given that such policies are used when streaming through large files and all tend to behave similarly under such conditions.

### 2.2.3 Probing File-Buffer Contents

To determine the state of the buffer cache after the eviction scan, we perform several probes, measuring the time to read one byte from selected pages. If the read call returns quickly, we assume the block of the file was resident in the cache; if the read returns slowly, we assume that a disk access was required. As noted elsewhere [3], it is not possible to perform a probe of every block to determine its state since this changes the state of the buffer cache; specifically, if *Dust* probes a block that was on disk, then this block will replace a block previously in the buffer cache, changing its state. Thus, we perform probes selectively.

To obtain an appropriate number of samples, we probe each stripe two times, for a total of twenty probes. The probes are spaced evenly across the test region, but the location of the first is chosen randomly from the first half of the first stripe. By keeping the probes relatively far apart, we ensure that they do not interfere with a later probe due to prefetching. Choosing a random offset for the probes allows one to run the benchmark multiple times to generate a better picture of the cache state. By running *Dust* multiple times on a platform, one is then able to accurately determine how the cache replacement policy chooses victim pages based on initial access, recency of access, and frequency of access.

<sup>1</sup>Precisely, the size of the eviction scan is set equal to the difference between the size of the cache and the size of the test region (i.e.,  $0.1 * \text{cache size}$ ) plus one half the size of the cache.

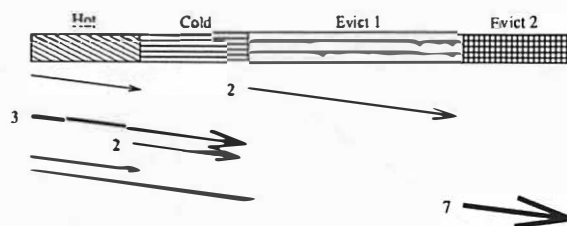


Figure 2: Access Pattern to Fingerprint History. Four distinct regions of file blocks (i.e., hot, cold, evict1, and evict2) are accessed to set attributes and cause evictions in order to identify whether or not history is being used by the replacement algorithm. Each arrow indicates a region that is being accessed; reads later in time move down the page. The width of each arrow along with a number, shows the number of times each block is read to set the frequency attributes.

## 2.3 Fingerprinting History

The fingerprinting tool described thus far can identify replacement policies containing a single queue ranking blocks based upon the three attributes. However, the previous step controls only the short-term attributes of blocks and thus cannot identify algorithms that track references to blocks that are no longer in memory (e.g., 2Q [12]) or that track the recency of references other than the last reference to each block (e.g., LRU-K [19]). To determine if long-term tracking is performed, *Dust* observes if preference is given to pages that have been referenced and then evicted before.

We now describe how the use of long-term history is identified. As shown in Figure 2, there are four regions of file blocks that are now accessed. The test region is now divided into two separate regions that are one half the total cache size, a *hot* and a *cold* portion. The algorithm begins by touching all of the hot pages and then evicting them by twice touching the *evict1* region; the *evict1* region contains sufficient blocks to entirely fill the buffer cache. Thus, the hot pages are no longer in the cache, but historical information about them is now tracked. *Dust* then touches the *hot* and *cold* regions three times and then touches *cold* two more times. At this point, *evict1* has been evicted entirely and *cold* is preferred whether initial access, recency or frequency attributes are being used by the replacement policy. Then *cold* is touched twice. This causes the *cold* region to be preferred by traditional LRU and LFU. *Hot* is then retouched, this additional reference gives the *hot* region preference in policies which use history. The last step prior to eviction is to rereference both the *hot* and *cold* regions sequentially. Notice that at this point the *hot* region has been touched the same number of times as the *cold* region but, it has been touched in such a way that it will have migrated into the long-term queue of a 2Q or LRU-2 cache, while the *cold* region will have not.

As in the short-term fingerprint, the next phase of *Dust* is to probe the test region to determine which blocks

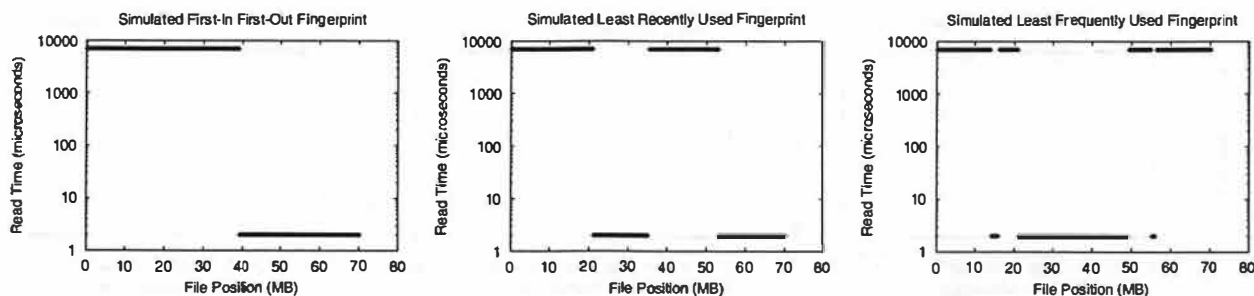


Figure 3: **Fingerprints of Basic Replacement Policies (FIFO, LRU, LFU).** The three graphs show the time required to probe blocks within the test region of a file depending upon the buffer cache replacement policy. The x-axis shows the offset of the probed block. The y-axis shows the time required for that probe; where low times ( $2\mu s$ ) indicate the block was in cache, whereas high times ( $7ms$ ) indicate the block was not in cache. From left to right, the graphs simulate FIFO, LRU, and LFU.

have been kept in the file cache. If the hot region remains in the cache, then we infer that history is being used. If the cold region remains in the cache, then we infer that history is not being used. Given that further identification of history attributes is likely to be specific to each replacement algorithm, we focus on only this simple historical fingerprint.

### 3 Simulation Fingerprints

To illustrate the ability of *Dust* to accurately fingerprint a variety of cache replacement policies, we have implemented a simple buffer cache simulator. In this section, we describe our simulation framework and then present a number of results. Our first simulation results verify the distinctive short-term replacement fingerprints produced for the pure replacement policies of FIFO, LRU, and LFU [23], as well as for other simple replacement policies such as Random and Segmented FIFO [31]. To explore the impact of internal state within the replacement policy, we investigate Clock [18] and Two-handed Clock [32]. We then demonstrate our ability to identify the use of historical information in the replacement policy, focusing on 2Q [12] and LRU-K [19]. We conclude this section by showing that *Dust* is robust to some inaccuracy in its estimate of buffer-cache size.

#### 3.1 Simulation methodology

Given that our simulator is meant only to illustrate the ability of *Dust* to identify different OS buffer cache replacement policies, we keep the rest of the system as simple as possible. Specifically, we assume that the only process running is our fingerprinting software, and thus ignore irregularities due to scheduling interference. We currently model only a buffer cache of a fixed size and do not consider any contention with the virtual memory system. For most of our simulations, we model a buffer cache containing approximately 80 MB (or 20,000 4 KB pages). Finally, we assume that reads that hit in the file

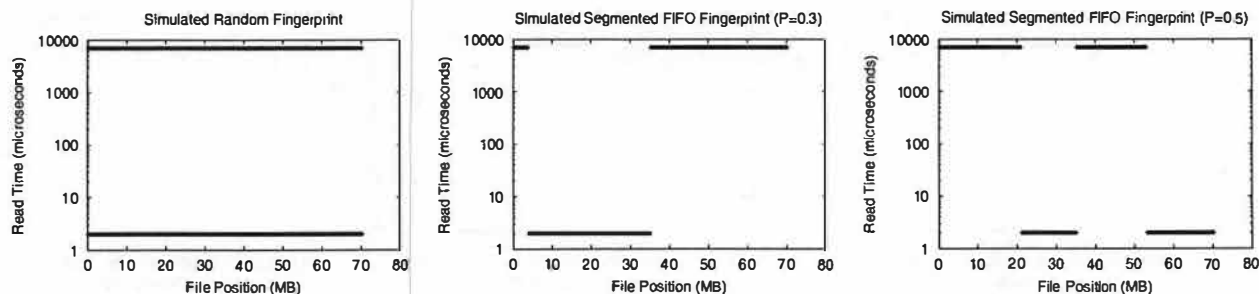
cache require a constant time of  $2\mu s$ , whereas reads that must go to disk require  $7ms$ .

#### 3.2 Basic Replacement Policies

We begin by showing that the simulation results for strict FIFO, LRU, and LFU replacement policies precisely matches what one can derive from the ordering graphs shown in Figure 1. The fingerprints from these three simulations are shown in Figure 3. We further show that *Dust* can identify Random replacement and Segmented FIFO [14]. These fingerprints are shown in Figure 4. Across all the graphs, one can observe the two levels of probe times, corresponding to blocks that are in cache and those that are not. Also, one can verify that approximately half of the test data remains in cache.

We now examine these basic policies in turn. The FIFO fingerprint shows that the second half of the test region remains in cache; this matches the initial access ordering shown in Figure 1 where blocks at the end of the file have priority. The LRU fingerprint shows that roughly the second quarter and the fourth quarter of the test region remains in the buffer cache; once again, this is the expected behavior since those blocks have been accessed the most recently. Finally, the LFU fingerprint shows that middle half of the file remains resident, as expected, since those blocks have the highest frequency counts. In the LFU fingerprint, one can see two small discontinuous regions that remain in cache to the left and right of the main in-cache area; this behavior is due to the fact that within each stripe, blocks have the same frequency count and these in-cache regions are part of a stripe that was beginning to be evicted.

Fingerprinting a Random replacement policy stresses the importance of running *Dust* multiple times. With a single fingerprint run of twenty probes, there exists some probability that Random replacement behaves identically to FIFO, LRU, or LFU. Therefore, by fingerprinting the system many times, we can definitively see that random pages are selected for replacement. This is illus-



**Figure 4: Fingerprints of Random and Segmented FIFO.** The left-most graph shows that a Random page replacement policy has a distinctive fingerprint; that each run of the fingerprint causes different pages to be evicted from the buffer cache. The middle graph shows Segmented FIFO with 30% of the buffer cache devoted to the secondary queue; the resulting fingerprint is a cyclic shift of the FIFO fingerprint. The right-most graph shows Segmented FIFO with at least 50% of the buffer cache devoted to the secondary queue; since this queue is managed with LRU, the fingerprint is identical to LRU.

trated in the first graph of Figure 4 with two horizontal lines indicating the “fast” and “slow” access times.

The original VMS system implemented the Segmented FIFO (SFIFO) page replacement policy [14]. SFIFO divides the buffer cache into two queues. The primary queue is managed by FIFO. Non-resident pages are faulted into the primary queue. When a page is evicted from the primary queue, it is moved to the secondary queue. If a page is accessed while in the secondary queue, it moves back into the primary queue. The key parameter in SFIFO is the fraction of the buffer cache devoted to the secondary queue, denoted  $P$  (thus,  $1 - P$  is the fraction devoted to the primary queue).

A value of  $P = 0.3$  is the traditional choice and is fingerprinted in the middle graph of Figure 4. The resulting SFIFO fingerprint is a cyclic shift of the pure FIFO fingerprint. The reason for this pattern is as follows. The initial read of the test area sets the contents of the primary and secondary queues such that the first pages accessed (*i.e.*, the left portion of the test area) are shifted down to the secondary queue and the tail of the primary queue; the right portion is at the head of the primary queue. When the pages are touched to set the recency and frequency attributes, the left portion of the test area is moved back to the head of the primary queue while the right portion is shifted down into the secondary queue and end of the primary queue. Thus, as blocks are evicted, the right portion is evicted first, followed by the first blocks of the left portion. Thus, with these queue sizes, SFIFO produces a distinctive fingerprint which can be used to uniquely identify this policy.

As  $P$  increases, SFIFO behaves more like LRU. When  $P \geq 0.5$  the fingerprint becomes identical to that of LRU, as shown in Figure 4. When the secondary queue is that large, by the time a page is touched for the second time, it has already progressed into the secondary queue. Thus, the fingerprint reveals the LRU behavior of the policy and matches the LRU fingerprint. We feel

that since Segmented FIFO is used to approximate LRU (especially with this high value of  $P$ ), it is acceptable, and even appropriate, that its fingerprint cannot be distinguished from that of LRU.

### 3.3 Replacement Policies with Initial State

The Clock replacement algorithm is a popular approach for managing unified file and virtual memory caches in modern operating systems, given its ability to approximate LRU replacement with a simpler implementation. The Clock algorithm is an interesting policy to fingerprint because it has two pieces of internal initial state: the initial position of the clock hand and whether or not each use bit is set. Thus, we must ensure that Clock can be identified by its fingerprint regardless of its initial state. We now describe small modifications to our methodology to guarantee this behavior.

In the basic implementation of Clock, the buffer cache is viewed as a circular buffer starting from the current position of the *clock hand*; a single *use bit* is associated with each page frame. Whenever a page is accessed, its use bit is set. When a replacement is needed, the clock hand cycles through page frames, looking for a frame with a cleared use bit and also clearing use bits as it inspects each frame. Thus, Clock approximates LRU by replacing pages that do not have their use bit set and have not been accessed for some time.

Since Clock treats the buffer cache as circular, the initial position of the clock hand does not affect our current fingerprint. The initial position of the clock hand simply determines where the first block of the test region is placed. Since all subsequent actions are relative to this initial position, this position is transparent to *Dust*. Thus, we do not need to modify our fingerprinting methodology to account for hand position.

However, the state of the use bits does impact our fingerprint. Depending upon the fraction of set use bits,  $U$ , the Clock fingerprint can look like FIFO or LRU. Specifically, when  $U$  is near the two extremes of 0 or 1, the

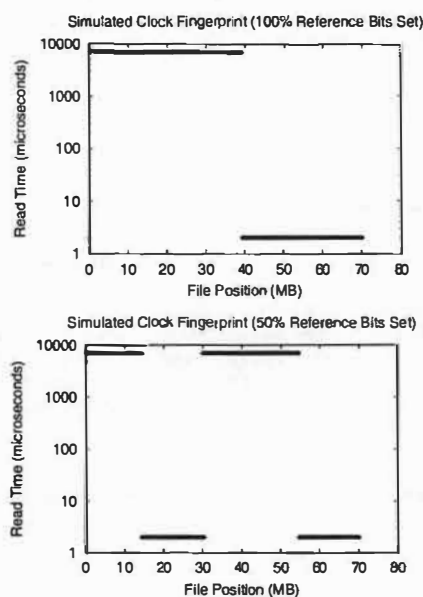


Figure 5: **Fingerprints of the Clock Replacement Policy.** To identify Clock, the basic fingerprinting algorithm is run twice. The first time it is run after the use bits have been all set; in this case, Clock behaves identically to FIFO as shown in the graph on the left. The second time it is run after half of the use bits have been set; in this case, Clock has the same fingerprint as LRU, as shown in the graph on the right.

fingerprint looks like FIFO; when  $U$  is near 0.5, the fingerprint looks like LRU. We now describe the intuition behind this behavior.

In the simplest case, when  $U = 0$ , each frame starting with the clock hand is allocated to sequential pages of the test region. As a result, the clock hand wraps back to the beginning of the buffer cache after this allocation and as *Dust* touches each page to set attributes, the use bit of every page is set. During eviction, the first pages of the test region are replaced, matching both the behavior and fingerprint of a FIFO policy. Note that  $U = 1$  results in identical behavior, except the clock hand must first sweep through all frames clearing use bits before it allocates the test region sequentially.

When  $U = 0.5$ , the left and right portions of the test region data are randomly interleaved in memory. This interleaving occurs because pages are allocated in two passes. In the first pass, those frames with cleared use bits are allocated to the left-hand portion of the test region; the use bits of these frames are then set and the use bits of the remaining frames are cleared. In the second pass, the remaining frames are allocated to the right-hand portion of the test region. In the accesses to set the locality and frequency attributes of the pages, the use bits of all frames are again set. Thus, when the eviction phase begins, the first half of pages from both the left and right portions of the test region are replaced. If

the frames with set use bits are uniformly distributed, this coincidentally matches the evictions of the LRU policy. If the distribution of use bits were not uniform, the fingerprint would show those blocks whose frames had their use bits initially clear as having been replaced. We consider the case where they are uniformly distributed as this provides a consistent and recognizable fingerprint.

Thus, to identify Clock, *Dust* brings the initial state of the use bits into each of these two configurations and observes the resulting two fingerprints. The following steps can be followed to configure the use bits from outside of the OS. *Dust* sets all of the use bits (*i.e.*,  $U = 1$ ) by allocating a *warmup* region of pages that fills the entire buffer cache and then touching all pages again (with no intervening allocations) so that their use bits are set.

Setting half of the use bits (*i.e.*,  $U = 0.5$ ) is slightly more complex. The first step is to set all the use bits as in the previous scenario. In the second step, *Dust* allocates a few more pages to the warmup region; since all of the reference bits are set at this point, the clock hand must pass through the entire buffer cache, clearing all of the reference bits, to find a page to evict. The final step is to randomly touch half of the pages, setting their use bits. In this way, *Dust* can configure the state of the use bits.

In summary, we modify *Dust* slightly to account for internal state. Before running any fingerprint, *Dust* first allocates the warmup region, which has the effect of setting use bits if the replacement policy implements them. If the resulting fingerprint looks like FIFO, then *Dust* runs again with half the use bits set. If the fingerprint still looks like FIFO, then we conclude that there are no use bits and the underlying policy is FIFO. If the second fingerprint looks like LRU, we conclude that Clock is the underlying policy. The result of running these two steps on the Clock replacement policy is shown in Figure 5.

### 3.4 Replacement Policies with History

We now show that *Dust* is able to distinguish those replacement policies that use long-term history from those that do not. We begin by briefly showing that the policies examined above (FIFO, LRU, LFU, Random, Segmented FIFO, and Clock) do not use history. We then discuss in more detail the behavior of those policies (LRU-K and 2Q) that do use history.

Figure 6 shows the long-term fingerprints of three representative policies that do not use history. The graph on the left is that for LRU; FIFO, LFU, and Segmented FIFO look identical and are not shown. The graph shows the results of probing the hot and cold regions of the test data. As expected, the hot data has been entirely evicted, as shown by its high probe times; although the initial portion of the cold data is also evicted due to the size of the eviction region, the cold data is clearly preferred by these policies. The middle graph shows that



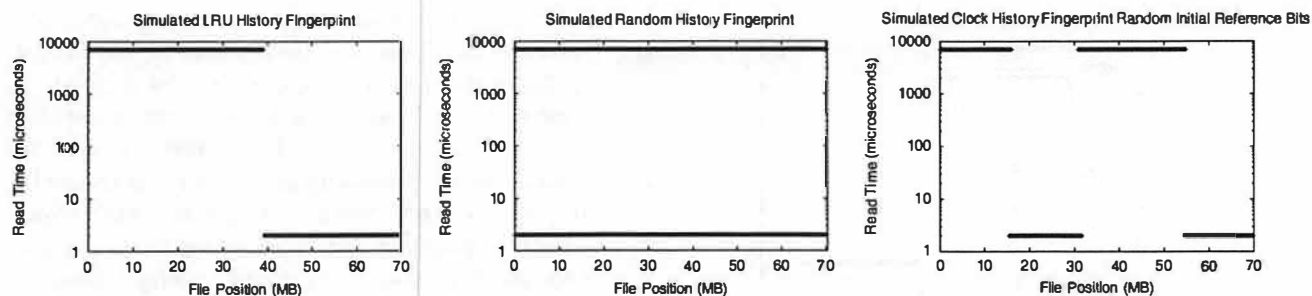


Figure 6: **History Fingerprint of Short-term Policies.** Probes are performed on only pages in the hot (i.e., the blocks on the left) and cold (i.e., the blocks on the right) test regions. The graph on the left shows the fingerprint for FIFO, LRU, LFU, and Segmented FIFO. Since the cold test region remains in the buffer cache, these policies do not prefer pages with history. The graph in the middle shows that Random also has no preference for pages with history and thus does not use history. Finally, the graph on the right shows that the historical fingerprint of Clock is ambiguous if the use bits are not set; after the use bits have been properly set, the fingerprint is identical to leftmost graph.

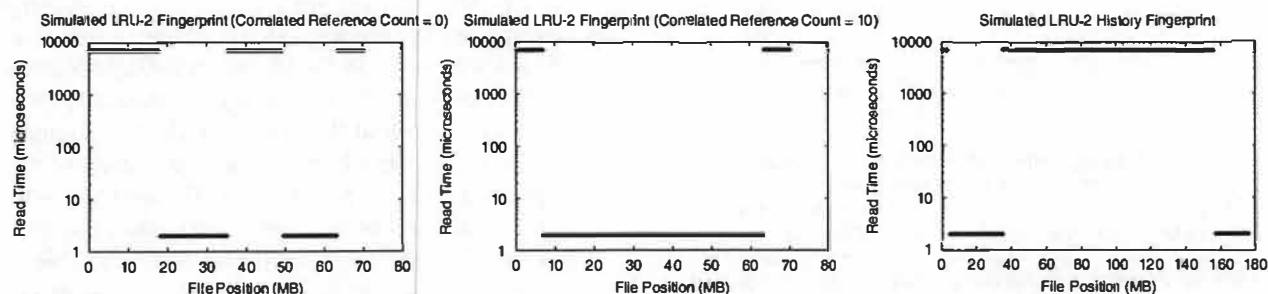


Figure 7: **Fingerprints of LRU-2.** The first graph shows the short-term fingerprint of LRU-2 when the correlated reference count is set to zero; in this case, LRU-2 displaces those pages with a frequency count less than 2 and those whose second-to-last reference is the oldest. The second graph shows the short-term fingerprint of LRU-2 when the correlated reference count is increased; here, no pages in the eviction with a frequency count higher than two are evicted. Finally, the last graph shows the history fingerprint of LRU-2, verifying that it prefers the hot pages.

Random has no preference for either hot or cold data. Finally, the graph on the right shows that the historical behavior of Clock is difficult to determine when the use bits are not explicitly controlled. In this graph, the use bits are set to  $U = 0.5$ ; as a result, the hot and cold regions are interleaved in the file buffer and then each region is replaced sequentially. To illustrate that Clock does not use history, *Dust* must again ensure that the use bits are all first cleared (or set); with this initialization step, the history fingerprint of Clock is identical to the first graph in the figure. Thus, FIFO, LRU, LFU, Segmented FIFO, Random, and Clock do not use history in making replacements.

The LRU-K replacement policy was introduced by the database community to address the problem that LRU is not able to discriminate between frequently and infrequently accessed pages [19]. The idea behind LRU-K is that it tracks the  $K$ -th reference to each page in the past, and replaces the page with the oldest  $K$ -th reference (or a page that does not have a  $K$ -th reference); thus, traditional LRU is equivalent to LRU-1. Given that  $K = 2$  exhibits most of the benefits of the general case, and is the most commonly used value, we only consider

LRU-2 further. LRU-2 is sensitive to another parameter as well, the correlated reference period,  $C$ ; the intuition is that accesses to a page within this period should not be counted as distinct references. Since setting  $C$  correctly is a non-trivial task, the default value for  $C$  is zero. Given that LRU-2 is complex, we note that our implementation is derived from the version provided by the original authors [20].

We begin by briefly exploring the sensitivity of LRU-2 to the correlated reference period; the short-term fingerprints of LRU-2 are shown in the first two graphs of Figure 7. When  $C = 0$  (i.e., the default value) the resulting fingerprint is a variation of pure LRU, as shown in the left-most graph. Specifically, the last stripe of the test region is evicted with LRU-2; since this stripe was accessed only twice, its second-to-last reference is very old (i.e., when the page was initially referenced). As the correlated reference period is increased such that  $C > 0$ , the fingerprint looks more similar to LFU, as shown in the middle graph. With this setting, pages in the eviction region are classified as having only correlated references and thus replace mostly themselves; thus, all of those pages that have a frequency count greater than two

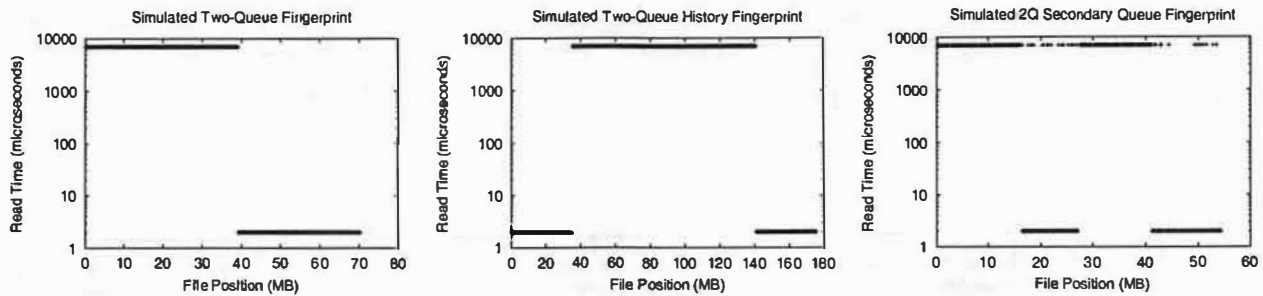


Figure 8: **Fingerprints of 2Q.** The first fingerprint of 2Q shows that the short-term replacement policy used is FIFO. The second fingerprint shows that 2Q uses history, preferring pages that have been accessed and then evicted. The third fingerprint shows that the replacement policy used for pages in the main queue is LRU.

are kept in memory. Finally, when  $C$  is very large, all accesses are treated as correlated and thus no pages have a second-to-last reference; in this case the behavior degenerates to pure LRU (not shown). In summary, LRU-2 produces a distinctive fingerprint that uniquely identifies it and also indicates the approximate setting of the correlated reference period.

Next, we verify that LRU-2 uses history. The last graph in Figure 7 shows the historical fingerprint of LRU-2. As desired, the hot region is given preference over data in the cold region; this occurs because the second-to-last reference of pages in the hot region is more recent than the second-to-last reference to those in the cold region. Further, when a replacement must be made within the hot region, those with the oldest second-to-last reference are chosen.

The 2Q algorithm was proposed as a simplification to LRU-2 with less run-time overhead yet similar performance [12]. The basic intuition behind 2Q is that instead of removing cold pages from the main buffer, it only admits hot pages to the main buffer. Thus, the buffer cache is divided into two buffers, a temporary queue for short-term accesses,  $A_{in}$  which is managed with FIFO, and the main buffer,  $A_m$ , which is managed with LRU. Pages are initially admitted into the  $A_{in}$  queue and only after they have been evicted and reaccessed are they admitted into  $A_m$ . Thus, 2Q has another structure to remember the pages that have been accessed but are no longer in the buffer cache,  $A_{out}$ . In our experiments, we set  $A_{in}$  to use 25% of the buffer cache (with  $A_m$  using the other 75%);  $A_{out}$  is able to remember a number of past references equal to 50% of the number of pages in the cache.

We show the fingerprints for 2Q in Figure 8. The first graph shows that the short-term fingerprint of 2Q is identical to FIFO. Given that the  $A_{in}$  queue is managed with FIFO and the short-term fingerprint does not access pages after they have been evicted, this is the expected result. However, 2Q can be easily distinguished from pure FIFO from observing the history fingerprint

shown in the second graph. In the historical fingerprint, we can see that the hot region remains entirely in the buffer cache, since these are the only accesses that are moved to the  $A_m$  buffer. Finally, we are able to identify the replacement policy employed by the long-term buffer,  $A_m$ , by setting the initial access, recency, and frequency attributes of the hot region and then forcing evictions from it. Since this methodology is more specific to the 2Q replacement policy, we do not describe it in more detail. This fingerprint is shown as the last graph of Figure 8 and correctly identifies the LRU policy of the  $A_m$  buffer. We note that for LRU-2 or other policies that use history, a similar technique could be used to determine the replacement strategy of the long-term queue. However, explicitly setting the state of the long-term queue requires knowledge of the policy of the short-term queue and the policy for moving a block from one queue to the other. Hence a fingerprinting technique for the long-term queue is by nature specific to the policy of the short-term queue.

### 3.5 Sensitivity to Buffer Size Estimate

In our last set of experiments we verify the robustness of *Dust* to inaccuracies in its estimate of the size of the buffer cache. If the estimate of the buffer cache size is significantly different than its actual value, then the resulting fingerprints are not identifiable. If the estimate of the cache is much too small, then *Dust* does not touch enough pages to force evictions to occur; if the estimate is much too large, then *Dust* evicts the entire region.

The short-term fingerprint is more sensitive to this estimate than the historical fingerprint: in the short-term fingerprint we must observe the presence or absence of stripes that use only 1/10th of the buffer cache, whereas in the historical fingerprint we must observe a hot or cold region that uses half of the buffer cache. However, as Figure 9 shows, the short-term fingerprint of LRU is distinguishable even with estimates that are either 20% under or over the real sizes. The other replacement policies, with the exception of Clock, are robust to a similar



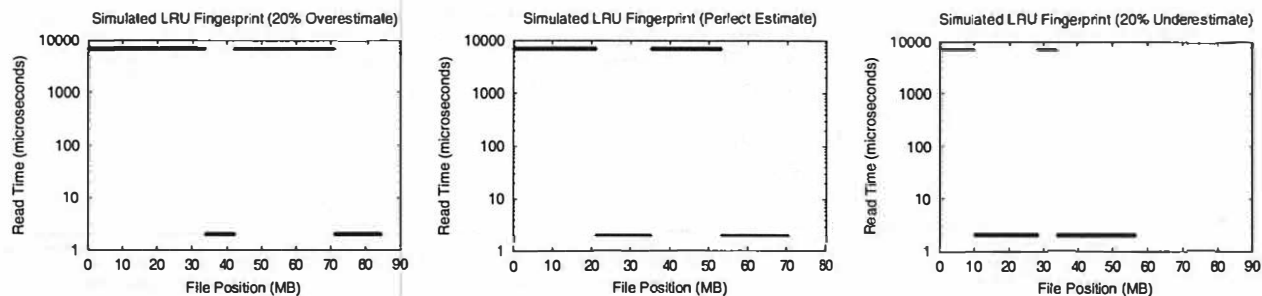


Figure 9: **Sensitivity of LRU Fingerprint to Cache Size Estimate.** These graphs show the short-term fingerprints of LRU as the estimate of the size of the buffer cache is varied. In the first graph the estimate is too high by 20%, in the second graph the estimate is perfect, and in the third graph the estimate is too low by 20%. However, all fingerprints still uniquely identify LRU.

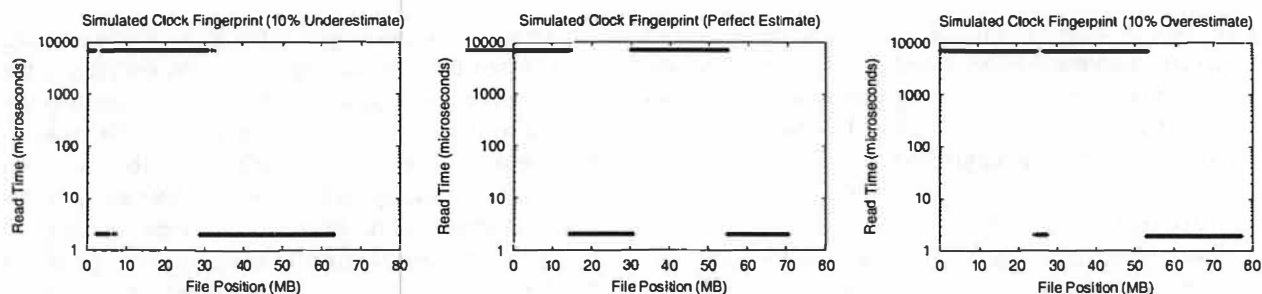


Figure 10: **Sensitivity of Clock Fingerprint to Cache Size Estimate.** These graphs show the short-term fingerprints of Clock with half of the use bits set as the estimate of the size of the buffer cache is varied. With  $U = 0.5$ , Clock is expected to look like LRU. In the first graph the estimate is too high by 10%, in the second graph the estimate is perfect, and in the third graph the estimate is too low by 10%. Thus, the Clock fingerprint is not as robust to inaccuracies in this estimate as the other algorithms.

degree.

The Clock replacement algorithm is more sensitive to this estimate due to our need to configure the state of the use bits. Specifically, the size of the warm-up region used by *Dust* to fill the buffer cache must be accurate as well. Figure 10 shows that *Dust* is still reasonably tolerant to errors in cache-size estimate when identifying Clock but not as robust as when identifying other algorithms.

## 4 Platform Fingerprints

Buffer caching in modern operating systems is often much more complex than the simple replacement policies described in operating systems textbooks. Part of this complexity is due to the fact that the filesystem buffer cache is integrated with the virtual memory system in many current systems; thus the amount of memory dedicated to the buffer cache can change dynamically based on the current workload. To control this effect, *Dust* minimizes the amount of virtual memory that it uses, and thus tries to maximize the amount of memory devoted to the file buffer cache. Further, we run *Dust* on an otherwise idle system to minimize disturbances from competing processes.

In this section, we describe our experience fin-

gerprinting three Unix-based operating systems: NetBSD 1.5, Linux 2.2.19 and 2.4.14, and Solaris 2.7. As we will see, the fingerprints of real systems contain much more variation than those of our simulations. In addition to fingerprinting the replacement policy of the buffer cache, *Dust* also reveals the cost of a hit versus a miss in the buffer cache, the size of the buffer cache, and whether or not the buffer cache is integrated with the virtual memory system.

*Dust* takes a considerable amount of time to run on a real system. Generating a sufficient number of data points requires running many iterations of test scan, eviction scan, and probes. In our experiments we always allowed at least 300 iterations. We found that one iteration can take anywhere from 30 seconds to three minutes depending on the system under test. Note that systems with smaller buffer caches can be tested in a shorter period of time since the test region becomes smaller. We feel this relatively long running time is acceptable since, for any given system configuration, *Dust* need only be run once; the results can be stored and made available to applications and programmers.

All of the experiments described in the section were run on systems with dual Pentium III-Xeon processors, 1 GB of physical RAM and a SCSI storage subsystem with Ultra2, 10000 RPM disks.

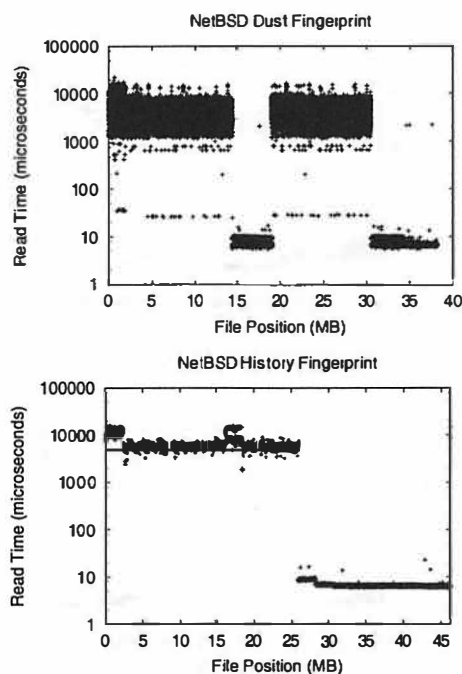


Figure 11: Fingerprints of NetBSD 1.5. The first graph shows the short-term fingerprint of NetBSD, indicating the LRU replacement policy. The second graph shows the long-term fingerprint, indicating that history is not used.

#### 4.1 NetBSD 1.5

Given that NetBSD 1.5 [16] has the most straightforward replacement policy of the systems we have examined, we begin with its fingerprint, shown in Figure 11. As in the simulations, we examine both short-term and long-term fingerprints. The first graph in Figure 11 shows the expected pattern for pure LRU replacement; given that *Dust* produces this same fingerprint regardless of whether it attempts to manipulate use bits, we can infer that NetBSD implements strict LRU, and not Clock. This conclusion is further verified by the second graph of Figure 11 showing that NetBSD does not use history. Documentation [16] and inspection of the source code [17] confirm our finding.

From the fingerprints we can also infer other parameters. Specifically, we can see that the time for reading a byte from a page in the buffer cache is on the order of  $10 \mu s$ , whereas the time for going to disk varies between about  $1 ms$  and  $10 ms$ . Further, even on this machine with 1 GB of physical memory, NetBSD devotes only about 50 MB to the buffer cache (most easily shown by the fact that the history fingerprint devotes this much memory to the hot and cold regions); this allows us to infer that the file buffer cache is segregated from the VM system.

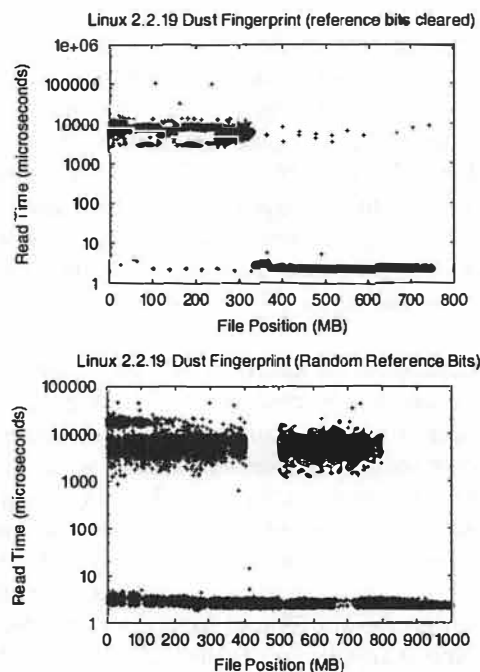


Figure 12: Fingerprints of Linux 2.2.19. The first graph shows the short-term fingerprint of Linux 2.2.19 when the use bits are all set; the second graph shows the fingerprint when the use bits are untouched.

#### 4.2 Linux 2.2.19

Linux 2.2.19 is a very popular version of the Linux kernel in production environments. In Section 5 we will run the NeST web server on top of this OS; thus, it is important for us to understand this fingerprint.

The short-term fingerprint of Linux 2.2.19 is shown in Figure 12. The graph on the left shows the results when *Dust* attempts to set all of the use bits. Since this graph looks like FIFO, we must investigate further to determine if Clock is actually being used. The graph on the right shows the fingerprint when the use bits are left in a random state. Although this fingerprint is very noisy, one can see that priority is given to pages that are most recently referenced (*i.e.*, pages near the second and fourth quarters); further, after filtering the data, we are able to verify that more pages in the first and third quarters are out of cache than in cache. Thus, this fingerprint is similar to the LRU fingerprint expected for a Clock-based replacement algorithm. Examination of the source code and documentation confirms that the replacement policy is Clock based [15, 34]. Finally, since the buffer cache size is very close to the amount of physical RAM in the system, we conclude a buffer cache that is integrated with the VM.

### 4.3 Linux 2.4.14

The memory management system within Linux underwent a large revision between version 2.2 and 2.4, thus we see a very different fingerprint for Linux 2.4.14, which uses a more complex replacement scheme than either Linux 2.2.19 or NetBSD. The short-term fingerprint, shown as the first graph in Figure 13, suggests that Linux 2.4 uses both a recency and frequency component, and does not use Clock. Further, the second graph of *Dust* shows that Linux 2.4 does use history in its decision.

Examination of the Linux 2.4.14 source code and existing documentation confirms these results [15, 34]. Linux maintains two separate queues: an active and an inactive list. When memory becomes scarce, Linux shrinks the size of the buffer cache. In doing this, pages that have not been recently referenced (as indicated by their reference bit) are moved from an active list to an inactive list. The inactive list is scanned for replacement victims using a form of page aging, in which an *age* counter is kept for each frame, indicating how desirable it is to keep this page in memory. When scanning for a page to evict, the page age is decreased as it is considered for eviction; when the page age reaches zero, the page is a candidate for eviction. The *age* is incremented whenever the page is referenced.

### 4.4 Solaris 2.7

Solaris presented us with the greatest challenge of the platforms we studied. The VM subsystem of Solaris has not been thoroughly studied; it is believed to use a two-handed, global Clock algorithm [7], but some researchers have noted non-intuitive behavior [3]. In two-handed Clock, one hand clears reference bits while the second hand follows some fixed distance behind, selecting a page for replacement if its reference bit is still clear. The hands are advanced in unison such that once the reference bit on a page is cleared, it has some opportunity to be re-referenced before it is a candidate for eviction. When implemented in our simulator, the fingerprint of two-handed Clock looks identical to FIFO (not shown).

The short-term fingerprint of Solaris 2.7 is shown in the first graph of Figure 14. The out-of-cache areas on both the far right and left of the fingerprint strongly suggests that Solaris is using a frequency (or aging) component in its eviction decision in addition to Clock. The second graph of Figure 14 shows the historical fingerprint for Solaris. Though the data is again noisy, it shows a clear preference for the hot region, again suggesting that history or page aging is also used in Solaris. The fingerprint also shows that the time to service a buffer cache hit is significantly higher in Solaris than in Linux. The fingerprint shows a hit time of over 10  $\mu$ s, whereas

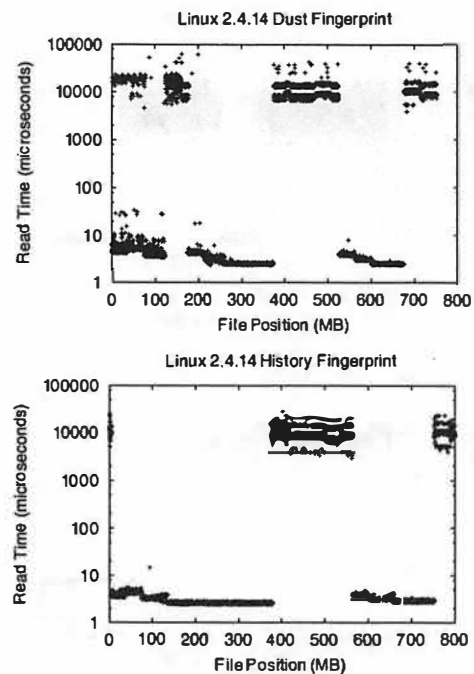


Figure 13: **Fingerprints of Linux 2.4.14.** The first graph shows the short-term fingerprint of Linux 2.4.14, indicating that a combination of LRU and LFU is used. The second graph shows the long-term fingerprint, indicating that history is used.

the hit time for Linux 2.4 on the same platform is under 10  $\mu$ s.

## 5 Cache-Aware Web Server

In this section, we describe how knowledge of the buffer cache replacement algorithm can be exploited to improve the performance of a real application. We do so by modifying a web server to re-order its accesses to first serve requests that are likely to hit in the file system cache, and only then serve those that are likely to miss. This idea of handling requests in a non-FIFO service order is similar to that introduced in connection scheduling web servers [9]; however, whereas that work scheduled requests based upon the size of the request, we schedule based upon predicted cache content. As we will see, re-ordering based on cache content both lowers average response time (by emulating a shortest-job first scheduling discipline) and improves throughput (by reducing total disk traffic).

### 5.1 Approach

The key challenge in implementing the cache-aware server is to use our gray-box knowledge of the file caching algorithm to determine which files are in the cache. By keeping track of the file access stream being presented to the kernel, the web server can simulate

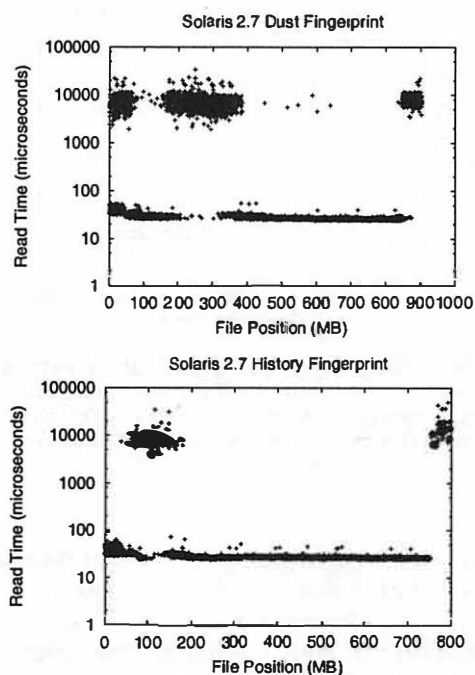


Figure 14: **Fingerprint of Solaris 2.7.** The first graph shows the short-term fingerprint of Solaris; the second graph shows the history fingerprint.

the operating system's buffer cache and thus predict at any given time what data is in cache. We term this *algorithmic mirroring*, and believe that it is a general and powerful manner in which to exploit gray-box knowledge.

One important assumption of algorithmic mirroring is that the application induces most or all of the traffic to the file system, and thus the mirror cache is likely to accurately represent the state of the real OS cache. Although this assumption may not hold in the general case within a multi-application environment, we believe it is feasible when a single application dominates all file-system activity. Server applications such as a web server or database management system are thus a perfect match for such mirroring methods.

The NeST storage appliance [6] supports HTTP as one of its many access protocols. NeST allows a configurable number of requests to be serviced simultaneously. Any requests received beyond that number are queued until one of the pending requests completes. By default, NeST services queued requests in FIFO order. We term this default behavior as *cache-oblivious NeST*.

We have modified the NeST request scheduler to keep a model of the current state of the OS buffer cache. The model is updated each time a request is scheduled. NeST bases its model of the underlying file cache on the algorithm exposed by *Dust*. NeST uses this model to reorder

requests such that those requests for files believed to be in cache are serviced first. Note that NeST does not perform caching of files itself, but relies strictly upon the OS buffer cache.

For the cache mirror to accurately reflect the internal state of the OS, NeST must have a reasonable estimate of the cache size. In our current approach, NeST uses the static estimate produced by *Dust*; the disadvantage of this approach is that this estimate is produced without contention with the virtual memory system, and thus may be larger than the amount available when the web server is actually running. To increase the robustness of our estimate, we plan to modify NeST to dynamically estimate the size of the buffer cache by measuring the time for each file access. If the time is "low", the file must have been in the cache, and if it is "high", the file was likely on disk. By comparing these timings with the prediction provided by the mirror cache, NeST can adjust the size of the mirror cache.

## 5.2 Performance

To evaluate the performance benefits of cache-aware scheduling, we compare the performance of cache-aware NeST to cache-oblivious NeST for two different workloads. In all tests, the web server is run on a dual Pentium III-Xeon machine with 128 MB of main memory and Ultra II disks. For clients, we use four machines (identical to the server, except containing 1 GB of main memory) each running 36 client threads. The clients are connected to the server with Gigabit Ethernet.

The server and clients are running Linux 2.2.19, which was shown in Section 4.2 to use the Clock replacement algorithm; therefore, cache-aware NeST is configured to model the Clock algorithm as well. In our configuration, the server has approximately 80 MB of memory dedicated to the buffer cache. In our experiments, we explore the performance of cache-aware NeST as we vary its estimate of the size of the buffer cache.

In our first experiment, we consider a workload in which each client thread repeatedly requests a random file from a set of 200 1 MB files. Figures 15 and 16 show the average response time and throughput, respectively for three different web servers: the Apache web server [1], cache-oblivious NeST, and cache-aware NeST as a function of its estimate of cache-size. We begin by comparing the response time and the throughput of NeST and Apache; from the two figures, we see that although NeST incurs some overhead for its flexible structure (*e.g.*, NeST can handle multiple transfer protocols, such as FTP and NFS), it achieves respectable performance as a web server and is a reasonable platform for studying cache-aware scheduling. Second, and most importantly, adding cache-aware scheduling signif-

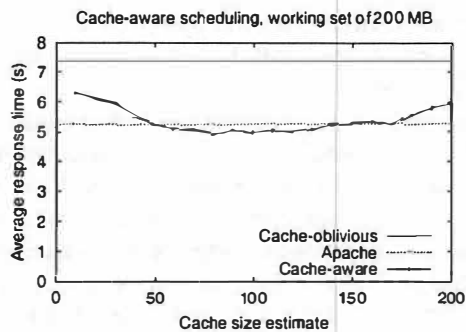


Figure 15: **Response Time as a Function of Cache Size Estimate.** Response time in cache-aware NeST is lowest when the estimate of cache size is closest to the true size of the cache.

icantly improves both the response time and the throughput of NeST. By first servicing requests that hit in the cache, cache-aware scheduling improves average response time by servicing short requests first. More dramatically, cache-aware scheduling improves throughput by reducing the number of disk reads (verified through the `/proc` interface): in-cache requests are handled before their data is evicted from the cache. Finally, the performance of cache-aware NeST improves when its estimate of the cache size is closer to the real value, but is robust to a large range of cache size estimates.

In our second experiment, we consider a workload created by the SURGE HTTP workload generator [5]. The SURGE workload uses approximately 12,000 distinct files with sizes taken from a Zipf distribution with a mean of approximately 21 KB. SURGE is thus a more representative web workload than is presented above.

With the SURGE workload, we measure qualitatively similar results to those above, except with two main differences. First, the performance of cache-oblivious NeST relative to Apache degrades slightly more; for example, the average response time for cache-oblivious NeST is 0.80 seconds and for Apache is 0.65 seconds. This result is expected, given that NeST is designed for staging data in the Grid, and is thus optimized for large files and not the small files more typical in web workloads. Second, the performance of cache-aware NeST is not as sensitive to its estimate of the cache size; for example, performance improves from 4.27 MB/s to 4.69 MB/s (approximately 10%) as the cache size estimate is improved from 10 MB to 80 MB. Apache achieves 4.91 MB/s. In the future, we plan to experiment with other web servers and workloads.

## 6 Related Work

The idea of using algorithmic knowledge of the underlying operating system to improve performance has been

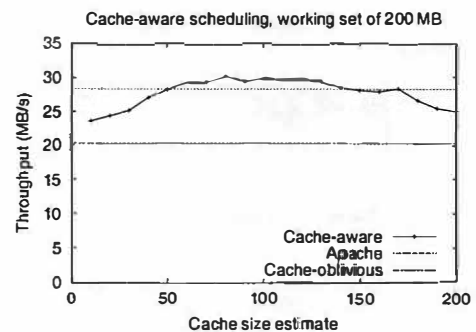


Figure 16: **Sensitivity to Cache Estimate Accuracy.** The performance of cache aware NeST improves as the estimate of cache size approaches the true size of the buffer cache. The buffer cache is approximately 80 MB. Cache-oblivious NeST and Apache are shown for comparison.

recently explored in the context of *gray-box systems* [3]. This work showed that an “OS-like” service can be implemented as an *Information and Control Layer (ICL)* outside of the OS, given algorithmic knowledge of the OS, probes of the OS, and statistical analysis. However, no concrete solutions were proposed for how developers of ICLs can obtain this algorithmic knowledge. In this paper, we show that fingerprinting can obtain this gray-box knowledge in a simple and automatic manner.

Fingerprinting system components to determine their behavior is not new and has been used successfully in other contexts, notably in networking and storage. Specifically, fingerprinting has been used to uncover key parameters within the TCP protocol and to identify the likely OS of a remote host [11, 21]. The primary difference between fingerprinting within TCP and in our context is that we are trying to identify policies that can have arbitrary behavior, rather than implementations that are expected to adhere to given specifications. In [25, 35] techniques similar to those used in *Dust* were used to determine various characteristics of disks, such as size of the prefetch window, prefetching algorithm and caching policy.

Fingerprinting also shares much in common with microbenchmarking. Specifically, both perform requests of the underlying system in order to characterize its behavior. For example, with simple probes in microbenchmarks, one can determine parameters of the memory hierarchy [2, 24], processor cycle time [28], and characteristics of disk geometry [26, 30]. In our view, the key difference between fingerprinting and microbenchmarking is that a fingerprint is used to discover the policy or algorithm employed by the underlying layer, whereas a microbenchmark is typically used to uncover specific system parameters.

The idea of discovering characteristics of lower layers



of a system and using that knowledge in higher layers to improve performance is not new. In traxtents [26] the file system layer of the operating system was modified to avoid crossing disk track boundaries so as to minimize the cost incurred due to head switching and exploit “zero-latency” access. Yu, *et al.* developed a method of predicting the position of the disk head without hardware support and used that information to determine which of several rotational replicas to use to service a given request [36], thus giving software expanded knowledge of hardware state.

Our approach involves informing the application of the buffer cache replacement policy in use by the operating system. SLEDs [33] and dynamic sets [29] seek to increase the knowledge that the application and operating system have of each other. Both take the approach of embellishing the interface between the OS and the application to allow the explicit exchange of certain types of information. In the case of dynamic sets, the application has the ability to provide more knowledge about its future access patterns. This allows the OS to reorder the fetching of data to improve cache performance. SLEDs allows the OS to export performance data to the application, enabling the application to modify its workload based on the performance characteristics of the underlying system.

The idea of servicing requests within a web server in a particular order was explored in connection-scheduling web servers [9]. The main thesis of that research is that better performance can be obtained by controlling the scheduling of requests within the web server, rather than with the OS. While their approach used static file size to schedule requests, cache-aware NeST uses a dynamic estimate of the contents of the buffer cache. In future work, we hope to investigate the interactions of scheduling requests based on both file size and cache content.

Our cache-aware web server has similarities to locality-aware request distribution (LARD) cluster-based web servers [22]. In LARD, the front-end node directs page requests to a specific back-end node based upon which back-end has most recently served this page (modulo load-balancing constraints); thus, the front-end has a simple model of the cache contents of each back-end and tries to improve their cache hit rates. Our approaches are complementary, as LARD partitions requests across different nodes, whereas we use cache content to service requests in a different order on a single node.

## 7 Conclusions and Future Work

We have shown that various buffer cache replacement algorithms can be uniquely identified with a simple fingerprint. Our fingerprinting tool, *Dust*, classifies al-

gorithms based upon whether they consider initial access, locality, frequency, and/or history when choosing a block to replace. With a simple simulator, we have shown that FIFO, LRU, LFU, Clock, Random, Segmented FIFO, 2Q, and LRU-K all produce distinctive fingerprints, allowing them to be uniquely identified. We have also begun to address the more challenging problem of fingerprinting real systems. By running *Dust* on NetBSD, Linux, and Solaris, we have shown that we can determine which attributes are considered by each page replacement algorithm. Finally, we have shown that the algorithmic knowledge revealed by *Dust* is useful for predicting the contents of the file cache. Specifically, we have implemented a cache-aware web server that services first those requests that are predicted to hit in the file cache, improving both response time and bandwidth.

In the near future, we would like to extend the range of policies which *Dust* is able to recognize. Specifically, we would like to see how *adaptive* policies such as EELRU [27] and LRFU [13] can be identified, as well as policies that use other attributes such as the size of a page or the cost of replacing a page. In our current system, one must visually interpret the fingerprint graphs produced by *Dust*; we would like to automate this process for the well-known replacement policies.

In the long-term, we plan to continue exploring fingerprinting of other subsystems within the OS (*e.g.*, the CPU scheduler). We would also like to determine how algorithmic knowledge can be used *across* several user processes; the main challenge is performing a model or simulation in which access to all OS inputs is not required for accuracy. Finally, we are investigating how algorithmic knowledge can be used not only to infer the contents of the file cache, but to change its contents as well.

## 8 Acknowledgments

We would like to thank Brian Forney, Tim Denehy, Muthian Sivathanu and Florentina Popovici for helpful discussion and comments on the paper. We would also like to thank our shepherd, Greg Ganger and the anonymous reviewers for their many helpful comments. This work is sponsored by NSF CCR-0092840, NGS-0103670, CCR-0133456, CCR-0098274, ITR-0086044, and the Wisconsin Alumni Research Foundation.

## References

- [1] Apache Foundation. Apache web server. <http://www.apache.org>.
- [2] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A

- Compiler Perspective. In *The 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, pages 320–331, Santa Margherita Ligure, Italy, June 1995.
- [3] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *The 18th Symposium on Operating Systems Principles (SOSP)*, October 2001.
  - [4] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-Performance Sorting on Networks of Workstations. In *SIGMOD '97*, Tucson, AZ, May 1997.
  - [5] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the SIGMETRICS '98 Conference*, June 1998.
  - [6] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Flexibility, Manageability, and Performance in a Grid Storage Appliance. In *To appear in HPDC-11*, 2002.
  - [7] J. L. Berton. Understanding solaris filesystems and paging. Technical Report TR-98-55, Sun Microsystems, 1998.
  - [8] P. Cao, E. W. Felten, and K. Li. Implementation and Performance of Application-Controlled File Caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 165–177, 1994.
  - [9] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection Scheduling in Web Servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.
  - [10] B. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Storage-Aware Caching: Revisiting Caching For Heterogeneous Storage Systems. In *The First USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, CA, January 2002.
  - [11] T. Glaser. TCP/IP Stack Fingerprinting Principles. [http://www.sans.org/newlook/resources/IDFAQ/TCP\\_fingerprinting.htm](http://www.sans.org/newlook/resources/IDFAQ/TCP_fingerprinting.htm), October 2000.
  - [12] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 439–450, September 1994.
  - [13] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On The Existence Of A Spectrum Of Policies That Subsumes The Least Recently Used (LRU) And Least Frequently Used (LFU) Policies. In *SIGMETRICS '99*, Atlanta, Georgia, May 1999.
  - [14] H. Levy and P. Lipman. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer*, 15(3):35–41, March 1982.
  - [15] Linux Kernel Archives. Linux source code. <http://www.kernel.org/>.
  - [16] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.
  - [17] NetBSD Kernel Archives. NetBSD 1.5 Source Code. <http://www.netbsd.org/>.
  - [18] V. F. Nicola, A. Dan, and D. M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *SIGMETRICS and PERFORMANCE*, 1992.
  - [19] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 297–306, 1993.
  - [20] P. O'Neil. Lru-2 source code. <ftp://ftp.cs.umb.edu/pub/lru-k/lru-k.tar.Z>.
  - [21] J. Padhye and S. Floyd. Identifying the TCP Behavior of Web Servers. In *SIGCOMM*, June 2001.
  - [22] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, 1998.
  - [23] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, 1990.
  - [24] R. H. Saavedra and A. J. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995.
  - [25] J. Schindler and G. R. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, 1999.
  - [26] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST)*, Monterey, CA, 2002.
  - [27] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Atlanta, GA, May 1999.
  - [28] C. Staelin and L. McVoy. mhz: Anatomy of a micro-benchmark. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 155–166, Berkeley, CA, June 1998.
  - [29] D. C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 252–263, Saint-Malo, France, October 1997.
  - [30] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.
  - [31] R. Turner and H. Levy. Segmented FIFO Page Replacement. In *1981 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1981.
  - [32] U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996.
  - [33] R. Van Meter and M. Gao. Latency Management in Storage Systems. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, October 2000.
  - [34] R. van Riel. Page replacement in linux 2.4 memory management. <http://www.surriel.com/lectures/linux24-vm.html>, June 2001.
  - [35] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *Proceedings of the 1995 ACM SIGMETRICS and PERFORMANCE Conference on Measurement and Modeling of Computer Systems*, pages 146–156, May 1995.
  - [36] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading Capacity for Performance in a Disk Array. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, CA, 2000.



# The JX Operating System

Michael Golm, Meik Felser, Christian Wawersich, Jürgen Kleinöder  
University of Erlangen-Nürnberg  
Dept. of Computer Science 4 (Distributed Systems and Operating Systems)  
Martensstr. 1, 91058 Erlangen, Germany  
{golm, felser, wawersich, kleinoeder}@informatik.uni-erlangen.de

## Abstract

*This paper describes the architecture and performance of the JX operating system. JX is both an operating system completely written in Java and a runtime system for Java applications.*

*Our work demonstrates that it is possible to build a complete operating system in Java, achieve a good performance, and still benefit from the modern software-technology of this object-oriented, type-safe language. We explain how an operating system can be structured that is no longer build on MMU protection but on type safety.*

*JX is based on a small microkernel which is responsible for system initialization, CPU context switching, and low-level protection-domain management. The Java code is organized in components, which are loaded into domains, verified, and translated to native code. Domains can be completely isolated from each other.*

*The JX architecture allows a wide range of system configurations, from fast and monolithic to very flexible, but slower configurations.*

*We compare the performance of JX with Linux by using two non-trivial operating system components: a file system and an NFS server. Furthermore we discuss the performance impact of several alternative system configurations. In a monolithic configuration JX achieves between about 40% and 100% Linux performance in the file system benchmark and about 80% in the NFS benchmark.*

## 1 Introduction

The world of software production has dramatically changed during the last decades from pure assembler programming to procedural programming to object-oriented programming. Each step raised the level of abstraction and increased programmer productivity. Operating systems, on the other hand, remained largely unaffected by this process. Although there have been attempts to build object-oriented or object-based operating systems (Spring [27], Choices [10], Clouds [17]) and many operating systems internally use object-oriented concepts, such as vnodes [31], there is a growing divergence between application programming and operating system programming. To close this semantic gap

between the applications and the OS interface a large market of middleware systems has emerged over the last years. While these systems hide the ancient nature of operating systems, they introduce many layers of indirection with several performance problems.

While previous object-oriented operating systems demonstrated that it is possible and beneficial to use object-orientation, they also made it apparent that it is a problem when implementation technology (object orientation) and protection mechanism (address spaces) mismatch. There are usually fine-grained “language objects” and large-grained “protected objects”. A well-known project that tried to solve this mismatch by providing object-based protection in hardware was the Intel iAPX/432 processor [37]. While this project is usually cited as a failure of object-based hardware protection, an analysis [14] showed that with a slightly more mature hardware and compiler technology the iAPX/432 would have achieved a good performance.

We believe that an operating system based on a dynamically compiled, object-oriented intermediate code, such as the Java bytecode, can outperform traditional systems, because of the many compiler optimizations (i) that are only possible at a late time (e.g., inlining virtual calls) and (ii) that can be applied only when the system environment is exactly known (e.g., cache optimizations [12]).

Using Java as the foundation of an operating system is attractive, because of its widespread use and features, such as interfaces, encapsulation of state, and automatic memory management, that raise the level of abstraction and help to build more robust software in less time.

To the best of our knowledge JX is the first Java operating system that has *all* of the following properties:

- The amount of C and assembler code is minimal to simplify the system and make it more robust.
- Operating system code and application code is separated in protection domains with strong isolation between the domains.
- The code is structured into components, which can be collocated in a single protection domain or dislocated in separate domains without touching the component code. This

reusability across configurations enables to adapt the system for its intended use, which may be, for example, an embedded system, desktop workstation, or server.

- Performance is in the 50% range of monolithic UNIX performance for computational-intensive OS operations. The difference becomes even smaller when I/O from a real device is involved.

Besides describing the JX system, the contribution of this paper consists of the first performance comparison between a Java OS and a traditional UNIX OS using real OS operations. We analyze two costs: (i) the cost of using a type-safe language, like Java, as an OS implementation language and (ii) the cost of extensibility.

The paper is structured as follows: In Section 2 we describe the architecture of the JX system and illustrate the cost of several features using micro benchmarks. Section 3 describes two application scenarios and their performance: a file system and an NFS server. Section 4 describes tuning and configuration options to refine the system and measures their effect on the performance of the file system. Section 5 concludes and gives directions for future research.

## 2 JX System Architecture

The majority of the JX system is written in Java. A small microkernel, written in C and assembler, contains the functionality that can not be provided at the Java level (system initialization after boot up, saving and restoring CPU state, low-level protection-domain management, and monitoring).

Figure 1 shows the overall structure of JX. The Java code is organized in components (Sec. 2.4) which are loaded into domains (Sec. 2.1), verified (Sec. 2.6), and translated to native code (Sec. 2.6). Domains encapsulate objects and threads. Communication between domains is handled by using portals (Sec. 2.2).

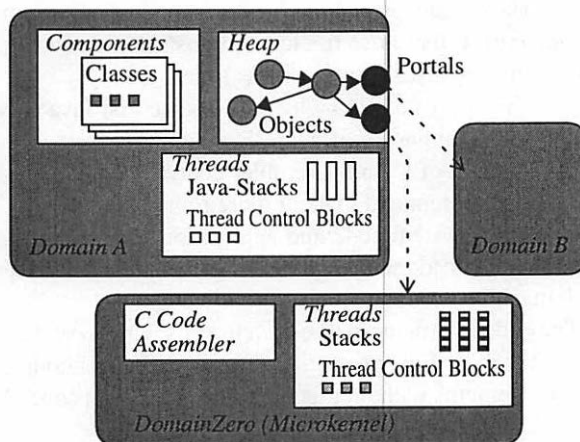


Figure 1: Structure of the JX system

The microkernel runs without any protection and therefore must be trusted. Furthermore, a few Java components must also be trusted: the code verifier, the code translator, and some hardware-dependent components (Sec. 2.7). These elements are the minimal *trusted computing base* [19] of our architecture.

### 2.1 Domains

The unit of protection and resource management is called a *domain*. All domains, except DomainZero, contain 100% Java code.

*DomainZero* contains all the native code of the JX microkernel. It is the only domain that can not be terminated. There are two ways how domains interact with DomainZero. First, explicitly by invoking services that are provided by DomainZero. One of these services is a simple name service, which can be used by other domains to export their services by name. Secondly, implicitly by requesting support from the Java runtime system; for example, to allocate an object or check a downcast.

Every domain has its own heap with its own garbage collector (GC). The collectors run independently and they can use different GC algorithms. Currently, domains can choose from two GC implementations: an exact, copying, non-generational GC or a compacting GC.

Every domain has its own threads. A thread does not migrate between domains during inter-domain communication. Memory for the thread control blocks and stacks is allocated from the domain's memory area.

Domains are allowed to share code - classes and interfaces - with other domains. But each domain has its own set of static fields, which, for example, allows each domain to have its own `System.out` stream.

### 2.2 Portals

Portals are the fundamental inter-domain communication mechanism. The portal mechanism works similar to Java's RMI [43], making it easy for a Java programmer to use it. A portal can be thought of as a proxy for an object that resides in another domain and is accessed using remote procedure call (RPC).

An entity that may be accessed from another domain is called *service*. A service consists of a normal object, which must implement a portal interface, an associated *service thread*, and an initial portal. A service is accessed via a *portal*, which is a remote (proxy) reference. Portals are capabilities [18] that can be copied between domains. The service holds a reference counter, which is incremented each time, the portal is duplicated. A domain that wants to offer a service to other domains can register the service's portal at a name server.

When a thread invokes a method at a portal, the thread is blocked and execution is continued in the service thread. All parameters are deep copied to the target domain. If a parameter is itself a portal, a duplicate of the portal is created in the target domain.

Copying parameters poses several problems. It leads to duplication of data, which is especially problematic when a large transitive closure is copied. To avoid that a domain is flooded with parameter objects, a per-call quota for parameter data is used in JX. Another problem is that the object identity is lost during copying. Although parameter copying can be avoided in a single addressspace system, and even for RPC between address spaces by using shared communication buffers [7], we believe that the advantages of copying outweigh its disadvantages. The essential advantage of copying is a nearly complete isolation of the two communicating protection domains. The only time where two domains can interfere with each other is during portal invocation. This makes it easy to control the security of the system and to restrict information flow. Another advantage of the copying semantics is, that it can be extended to a distributed system without much effort.

In practice, copying posed no severe performance problems, because only small data objects are used as parameters. Objects with a large transitive closure in most cases are server objects and are accessed using portals. Using them as data objects often is not intended by the programmer.

As an optimization the system checks whether the target domain of a portal call is identical to the current domain and executes the call as a function invocation without thread switch and parameter copy.

When a portal is passed as a parameter in a portal call, it is passed by-reference. As a convenience to the programmer the system also allows an object that implements a portal interface to be passed like a portal. First it is checked, whether this object already is associated with a service. In this case, the existing portal is passed. Otherwise, a service is launched by creating the appropriate data structures and starting a new service thread. This mechanism allows the programmer to completely ignore the issue of whether the call is crossing a domain border or not. When the call remains inside the domain the object is passed as a normal object reference. When the call leaves the domain, the object automatically is promoted to a service and a portal to this service is passed.

When a portal is passed to the domain in which its service resides, a reference to the service object is passed instead of the portal.

When two domains want to communicate via portals they *must* share some types. These are at least the portal interface and the parameter types. When a domain

System	IPC (cycles)
L4Ka (PIII, 450MHz) [32]	818
Fiasco/L4 (PIII 450 MHz) [42]	2610
J-Kernel (LRMI on MS-VM, PPro 200MHz) [28]	440
Alta/KaffeOS (PII 300 MHz) [5]	27270
JX (PIII 500MHz)	650

Table 1: IPC latency (round-trip, no parameters)

obtains a portal, it is checked whether the correct interface is present.

Each time a new portal to a service is created a reference counter in the service control block is incremented. It is decremented when a portal is collected as garbage or when the portal's domain terminates. When the count reaches zero the service is deactivated and all associated resources, such as the service thread, are released.

Table 1 shows the cost of a portal invocation and compares it with other systems. This table contains very different systems with very different IPC mechanisms and semantics. The J-Kernel IPC, for example, does not even include a thread switch.

**Fast portals.** Several portals which are exported by DomainZero are *fast portals*. A fast portal invocation looks like a normal portal invocation but is executed in the caller context (the caller thread) by using a function call - or even by inlining the code (see also Sec. 4.2.2). This is generally faster than a normal portal call, and in some cases it is even necessary. For example, DomainZero provides a portal with which the current thread can yield the processor. It would make no sense to implement this method using the normal portal invocation mechanism.

## 2.3 Memory objects

An operating system needs an abstraction to represent large amounts of memory. Java provides byte arrays for this purpose. However, arrays have several shortcomings, that make them nearly unsuitable for our purposes. They are not accessed using methods and thus the set of allowed operations is fixed. It is, for example, not possible to restrict access to a memory region to a read-only interface. Furthermore, arrays do not allow revocation and subrange creation - two operations that are essential to pass large memory chunks without copying.

To overcome these shortcomings we developed another abstraction to represent memory ranges: *memory objects*. Memory objects are accessed like normal objects via method invocations. But such invocations are treated specially by the translator: they are replaced by the machine instructions for the memory access. This makes memory access as fast as array access.

Memory objects can be passed between domains like portals. The memory that is represented by a memory object is not copied when the memory object is passed to another domain. This way, memory objects implement shared memory.

Access to a memory range can be revoked. For this purpose all memory portals that represent the same range of memory contain a reference to the same central data structure in DomainZero. Among other information this data structure contains a valid flag. The revocation method invalidates the original memory object by clearing the valid flag and returns a new one that represents the same range of memory. Memory is not copied during revocation but all memory portals that previously represented this memory become invalid.

When a memory object is passed to another domain, a reference counter, which is maintained for every memory range, is incremented. When a memory object - which, in fact, is a portal or proxy for the real memory - is garbage collected, the reference counter is decremented. This happens also for all memory objects of a domain that is terminated. To correct the reference counts the heap must be scanned for memory objects before it is released.

**ReadOnlyMemory.** ReadOnlyMemory is equivalent to Memory but it lacks all the methods that modify the memory. A ReadOnlyMemory object can not be converted to a Memory object.

**DeviceMemory.** DeviceMemory is different from Memory in that it is not backed by main memory: It is usually used to access the registers of a device or to access memory that is located on a device and mapped into the CPU's address space. The translator knows about this special use and does not reorder accesses to a DeviceMemory. When a DeviceMemory is garbage collected the memory is not released.

## 2.4 Components

All Java code that is loaded into a domain is organized in components. A component contains the classes, interfaces, and additional information; for example, about dependencies from other components or about the required scheduling environment (preemptive, nonpreemptive).

**Reusability.** An overall objective of object orientation and object-oriented operating systems is code reuse. JX has all the reusability benefits that come with object orientation. But there is an additional problem in an operating system: the protection boundary. To call a module across a protection boundary in most operating system is different from calling a module inside the own protection domain. Because this difference is a big hindrance on the

way to reusability, this problem has already been investigated in the microkernel context [22].

Our goal was a reuse of components in different configurations without code modifications. Although the portal mechanism was designed with this goal the programmer must keep several points in mind when using a portal. Depending on whether the called service is located inside the domain or in another domain there are a few differences in behavior. Inside a domain normal objects are passed by reference. When a domain border is crossed, parameters are passed by copy. To write code that works in both settings the programmer must not rely on either of these semantics. For example, a programmer relies on the reference semantics when modifying the parameter object to return information to the caller; and the programmer relies on the copy semantics when modifying the parameter object assuming this modification does not affect the caller.

In practice, these problems can be relieved to a certain extent by the automatic promotion of portal-capable objects to services as described in Section 2.2. By declaring all objects that are entry points into a component as portals a reference semantics is guaranteed for these objects.

**Dependencies.** Components may depend on other components. We say that component B has an *implementation dependence* on component A, if the method implementations of B use classes or interfaces from A. Component B has an *interface dependence* on component A if the method signatures of B use classes or interfaces from A or if a class/interface of B is a subclass/subinterface of a class/interface of A, or if a class of B implements an interface from A, or if a non-private field of a class of B has as its type a class/interface from A.

Component dependencies must be non-cyclic. This requirement makes it more difficult to split existing applications into components (Although they can be used as one component!). A cyclic dependency between components usually is a sign of bad design and should be removed anyway. When a cyclic dependency is present, it must be broken by changing the implementation of one component to use an interface from an unrelated component while the other class implements this interface. The components then both depend on the unrelated component but not on each other. The dependency check is performed by the verifier and translator.

We used Sun's JRE 1.3.1\_02 for Linux to obtain the transitive closure of the depends-on relation starting with java.lang.Object. The implementation dependency consists of 625 classes; the interface dependency consists of 25 classes. This means, that each component that uses the Object class (i.e., every component) depends on at least 25 classes from the JDK. We think, that even 25

classes are a too broad foundation for OS components and define a compatibility relation that allows to exchange the components.

**Compatibility.** The whole system is build out of components. It is necessary to be able to improve and extend one component without changing all components that depend on this component. Only a component B that is compatible to component A can be substituted for A. A component B is binary compatible to a component A, if

- for each class/interface  $C_A$  of A there is a corresponding class/interface  $C_B$  in component B
- class/interface  $C_B$  is binary compatible to class  $C_A$  according to the definition given in the “Java Language Specification” [26] Chapter 13.

When a binary compatible component is also a semantic superset of the original component, it can be substituted for the original component without affecting the functionality of the system.

**JDK.** The JDK is implemented as a normal component. Different implementations and versions can be used. Some classes of the JDK must access information that is only available in the runtime system. The class `Class` is an example. This information is obtained by using a portal to `DomainZero`. In other words, where a traditional JDK implementation would use a native method, JX uses a normal method that invokes a service of `DomainZero` via a portal. All of our current components use a JDK implementation that is a subset of a full JDK and, therefore, can also be used in a domain that loads a full JDK.

**Interface invocation.** Non-cyclic dependencies and the compilation of whole components opens up a way to compile very efficient interface invocations. Usually, interface invocations are a problem because it is not possible to use a fixed index into a method table to find the interface method. When different classes implement the interface, the method can be at different positions in their method tables. There exists some work to reduce the overhead in a system that does not impose our restrictions [1]. In our translator we use an approach that is similar to selector coloring [20]. It makes interface invocations as fast as method invocations at the cost of (considerably) larger method tables.

The size of the x86 machine code in the complete JX system is 1,010,752 bytes, which was translated from 230,421 bytes of bytecode. The method tables consume 630,388 bytes. These numbers show that it would be worthwhile to use a compression technique for the method tables or a completely different interface invocation mechanism. One should keep in mind, that a technique as described in [1] has an average-case performance near to a virtual invocation, but it may be difficult

to analyze the worst-case behavior of the resulting system, because of the use of a caching data structure.

## 2.5 Memory management

Protection is based on the use of a type-safe language. Thus an MMU is not necessary. The whole system, including all applications, runs in one physical address space. This makes the system ideally suited for small devices that lack an MMU. But it also leads to several problems. In a traditional system fragmentation is not an issue for the user-level memory allocator, because allocated, but unused memory, is paged to disk. In JX unused memory is wasted main memory. So we face a similar problem as kernel memory allocators in UNIX, where kernel memory usually also is not paged and therefore limited. In UNIX a kernel memory allocator is used for vnodes, proc structures, and other small objects. In contrast to this the JX kernel does not create many small objects. It allocates memory for a domain's heap and the small objects live in the heap. The heap is managed by a garbage collector. In other words, the JX memory management has two levels, a global management, which must cope with large objects and avoid fragmentation, and a domain-local garbage-collected memory. The global memory is managed using a bitmap allocator [46]. This allocator was easy to implement, it automatically joins free areas, and it has a very low memory footprint: Using 1024-byte blocks and managing about 128MBytes or 116977 blocks, the overhead is only 14622 bytes or 15 blocks or 0.01 percent. However, it should not be too complicated to use a different allocator.

To give up the MMU means that several of their responsibilities (besides protection) must be implemented in software. One example is the stack overflow detection, another one the null pointer detection. Stack overflow detection is implemented in JX by inserting a stack size check at the beginning of each method. This is feasible, because the required size of a stack frame is known before the method is executed. The size check has a reserve, in case the Java method must trap to a runtime function in `DomainZero`, such as `checkcast`. The null pointer check currently is implemented using the debug system of the Pentium processor. It can be programmed to raise an exception when data or code at address zero is accessed. On architectures that do not provide such a feature, the compiler inserts a null-pointer check before a reference is used.

A domain has two memory areas: an area where objects may be moved and an area where they are fixed. In the future, a single area may suffice, but then all data structures that are used by a domain must be movable. Currently, the fixed area contains the code and class information, the thread control blocks and stacks. Mov-



ing these objects requires an extension of the system: all pointers to these objects must be known to the GC and updated; for example, when moving a stack, the frame pointers must be adjusted.

## 2.6 Verifier and Translator

The verifier is an important part of JX. All code is verified before it is translated to native code and executed. The verifier first performs a standard bytecode verification [48]. It then verifies an upper limit for the execution times of the interrupt handlers and the scheduler methods (Sec. 2.8) [2].

The translator is responsible for translating byte code to machine code, which in our current system is x86 code. Machine code can either be allocated in the domain's fixed memory or in DomainZero's fixed memory. Installing it in DomainZero allows to share the code between domains.

## 2.7 Device Drivers

An investigation of the Linux kernel has shown that most bugs are found in device drivers [13]. Because device drivers will profit most from being written in a type-safe language, all JX device drivers are written in Java. They use DeviceMemory to access the registers of a device and the memory that is available on a device; for example, a frame buffer. On some architectures there are special instructions to access the I/O bus; for example, the in and out processor instructions of the x86. These instructions are available via a fast portal of DomainZero. As other fast portals, these invocations can be inlined by the translator.

**DMA.** Most drivers for high-throughput devices will use busmaster DMA to transfer data. These drivers, or at least the part that accesses the DMA hardware, must be trusted.

**Interrupts.** Using a portal of DomainZero, device drivers can register an object that contains a `handleInterrupt` method. An interrupt is handled by invoking the `handleInterrupt` method of the previously installed interrupt handler object. The method is executed in a dedicated thread while interrupts on the interrupted CPU are disabled. This would be called a *first-level interrupt handler* in a conventional operating system. To guarantee that the handler can not block the system forever, the verifier checks all classes that implement the `InterruptHandler` interface. It guarantees that the `handleInterrupt` method does not exceed a certain time limit. To avoid undecidable problems, only a simple code structure is allowed (linear code, loops with constant bound and no write access to the loop variable inside the loop). A `handleInterrupt` method usually acknowledges the interrupt at the

device and unblocks a thread that handles the interrupt asynchronously.

We do not allow device drivers to disable interrupts outside the interrupt handler. Drivers usually disable interrupts as a cheap way to avoid race conditions with the interrupt handler. Code that runs with interrupts disabled in a UNIX kernel is not allowed to block, as this would result in a deadlock. Using locks also is not an option, because the interrupt handler - running with interrupts disabled - should not block. We use the abstraction of an `AtomicVariable` to solve these problems. An `AtomicVariable` contains a value, that can be changed and accessed using `set` and `get` methods. Furthermore, it provides a method to atomically compare its value with a parameter and block if the values are equal. Another method atomically sets the value and unblocks a thread. To guarantee atomicity the implementation of `AtomicVariable` currently disables interrupts on a uniprocessor and uses spinlocks on a multiprocessor. Using `AtomicVariables` we implemented, for example, a producer/consumer list for the network protocol stack.

## 2.8 Scheduling

There is a common experience that the scheduler has a large impact on the system's performance. On the other hand, no single scheduler is perfect for all applications.

Instead of providing a configuration interface to the scheduler we follow our methodology of allowing a user to completely replace an implementation, in this case the scheduler. Each domain may also provide its own scheduler, optimized for its particular requirements.

The scheduler can be used in several configurations:

- First, there is a scheduler that is build into the kernel. This scheduler is only used for performance analysis, because it is written in C and can not be replaced at run time.
- The kernel can be compiled without the built-in scheduler. Then all scheduling decisions lead to the invocation of a scheduler implementation which is written in Java. In this configuration there is one (Java) scheduler that schedules all threads of all domains.
- The most common configuration, however, is a two-level scheduling. The global scheduler does not schedule threads, as in the previous configuration, but domains. Instead of activating an application thread, it activates the scheduler thread of a domain. This domain-local scheduler is responsible for selecting the next application thread to run. The global scheduler knows all domain-local schedulers and a domain-local scheduler has a portal to the global scheduler. On a multiprocessor there is one global scheduler per pro-

cessor and the domains possess a reference to the global schedulers of the processors on which they are allowed to run.

The global scheduler must be trusted by all domains. The global scheduler does not need to trust a domain-local scheduler. This means, that the global scheduler can *not* assume, that an invocation of the local scheduler returns after a certain time.

To prevent one domain monopolizing the processor, the computation can be interrupted by a timer interrupt. The timer interrupt leads to the invocation of the global scheduler. This scheduler first informs the scheduler of the interrupted domain about the pre-emption. It switches to the domain scheduler thread and invokes the scheduler's method `preempted()`. During the execution of this method the interrupts are disabled. An upper bound for the execution time of this method has been verified during the verification phase. When the method `preempted()` returns, the system switches back to the thread of the global scheduler. The global scheduler then decides, which domain to run next activates the domain-local scheduler using the method `activated()`. For each CPU that can be used by a domain the local scheduler of the domain has a CPU portal. It activates the next runnable thread by calling the method `switchTo()` at the CPU portal. The `switchTo()` method can only be called by a thread that runs on the CPU which is represented by the CPU portal. The global scheduler does not need to wait for the method `activated()` to finish. Thus, an upper time bound for method `activated()` is not necessary. This method makes the scheduling decision and it can be arbitrarily complex.

If a local scheduler needs smaller time-slices than the global scheduler, the local scheduler must be interrupted without being pre-empted. For this purpose, the local scheduler has a method `interrupted()` which is called before the time-slice is fully consumed. This method operates similar to the method `activated()`.

Because our scheduler is implemented outside the microkernel and there are operations of the microkernel that affect scheduling, for example, thread handoff during a portal invocation, we face a similar situation as a user-level thread implementation on a UNIX-like system. A well-known solution are scheduler activations [3], which notify the user-level scheduler about events inside the kernel, such as I/O operations. JX uses a similar approach, although there are very few scheduling related operations inside the kernel. Scheduling is affected when a portal method is invoked. First, the scheduler of the calling domain is informed, that one thread performs a portal call. The scheduler can now delay the portal call, if there is any other runnable thread in this domain. But it can as well handoff the processor to the target domain. The scheduler of the service domain

is notified of the incoming portal call and can either activate the service thread or let another thread of the domain run. Not being forced to schedule the service thread immediately is essential for the implementation of a non-preemptive domain-local scheduler.

This extra communication is not for free. The time of a portal call increases from 650 cycles (see Table 1) to 920-960 cycles if either the calling domain or the called domain is informed. If both involved domain schedulers are informed about the portal call the required time increases to 1180 cycles.

## 2.9 Locking and condition variables

**Kernel-level locking.** There are very few data structures that must be protected by locks inside `DomainZero`. Some of them are accessed by only one domain and can be locked by a domain-specific lock. Others, for example, the domain management data structures, need a global lock. Because the access to this data is very short, an implementation that disables interrupts on a uniprocessor and uses spinlocks on a multiprocessor is sufficient.

**Domain-level locking.** Domains are responsible for synchronizing access to objects by their own threads. Because there are no objects shared between domains there is no need for inter-domain locking of objects. Java provides two facilities for thread synchronization: mutex locks and condition variables. When translating a component to native code, an access to such a construct is redirected to a user-supplied synchronization class. How this class is implemented can be decided by the user. It can provide no locking at all or it can implement mutexes and condition variables by communicating with the (domain-local) scheduler. Every object can be used as a monitor (mutex lock), but very few actually are. To avoid allocating a monitor data structure for every object, traditional JVMs either use a hashtable to go from the object reference to the monitor or use an additional pointer in the object header. The hashtable variant is slow and is rarely used in today's JVMs. The additional pointer requires that the object layout must be changed and the object header be accessible to the locking system. Because the user can provide an own implementation, these two implementations, or a completely application-specific one, can be used.

**Inter-domain locking.** Memory objects allow sharing of data between domains. JX provides no special inter-domain locking mechanisms. When two domains want to synchronize, they can use a portal call. We did not need such a feature yet, because the code that passes memory between domains does it by explicitly revoking access to the memory.



### 3 Application Scenarios: Comparing JX to a Traditional Operating System

JX contains a file system component that is a port of the Linux ext2 file system to Java [45]. Figure 2 shows the configuration, where file system and buffer cache are cleanly separated into different components. The gray areas denote protection domains and the white boxes components. The file system uses the Buffer-Cache interface to access disk blocks. To read and write blocks to a disk the buffer cache implementation uses a reference to a device that implements the BlockIO interface. The file system and buffer cache components do not use locking. They require a non-preemptive scheduler to be installed in the domain.

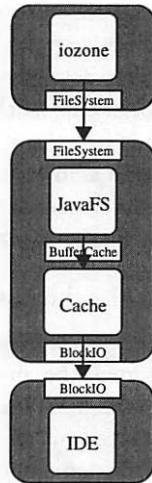


Figure 2: IOZone configuration

To evaluate the performance of JX we used two benchmarks: the IOZone benchmark [44] to assess file system performance and a home brewed *rate* benchmark to assess the performance of the network stack and NFS server. The rate benchmark sends *getattr* requests to the NFS server as fast as possible and measures the achievable request rate. As JX is a pure Java system, we can not use the original IOZone program, which is written in C. Thus we ported IOZone to Java. The JX results were obtained using our Java version and the Linux results were obtained using the original IOZone.

The hardware consists of the following components:

- The system-under-test: PIII 500MHz with 256 MBytes RAM and a 100 MBit/s 3C905B Ethernet card running Suse Linux 7.3 with kernel 2.4.0 or JX.

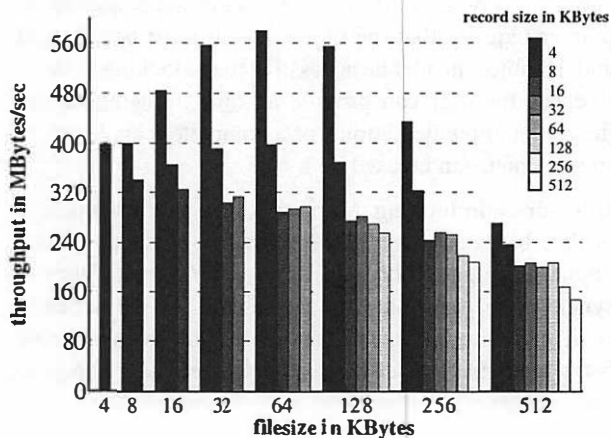


Figure 3: Linux IOZone performance

- The client for the NFS benchmark: a PIII 1GHz with a 100 MBit/s 3C905B Ethernet card running Suse Linux 7.3.
- A 100MBit/s hub that connects the two systems.

Figure 3 shows the results of running the IOZone reread benchmark on Linux.

Our Java port of the IOZone contains the write, rewrite, read, and reread parts of the original benchmark. In the following discussion we only use the reread part of the benchmark. The read benchmark measures the time to read a file by reading fixed-length records. The reread benchmark measures the time for a second read pass. When the file is smaller than the buffer cache all data comes from the cache. Once a disk access is involved, disk and PCI bus data transfer times dominate the result and no conclusions about the performance of JX can be drawn. To avoid these effects we only use the reread benchmark with a maximum file size of 512 KBytes, which means that the file completely fits into the buffer cache. The JX numbers are the mean of 50 runs of IOZone. The standard deviation was less than 3%. For time measurements on JX we used the Pentium timestamp counter which has a resolution of 2 ns on our system.

Figure 2 shows the configuration of the JX system when the IOZone benchmark is executed. Figure 4 shows the results of the benchmark. Figure 5 compares JX performance to the Linux performance. Most combinations of file size and record size give a performance between 20% and 50% of the Linux performance. Linux is especially good at reading a file using a small record size. The performance of this JX configuration is rather insensitive to the record size. We will explain how we improved the performance of JX in the next section.

Another benchmark is the rate benchmark, which measures the achievable NFS request rate by sending *getattr* requests to the NFS server. Figure 6 shows the domain structure of the NFS server: all components are placed in one domain, which is a typical configuration

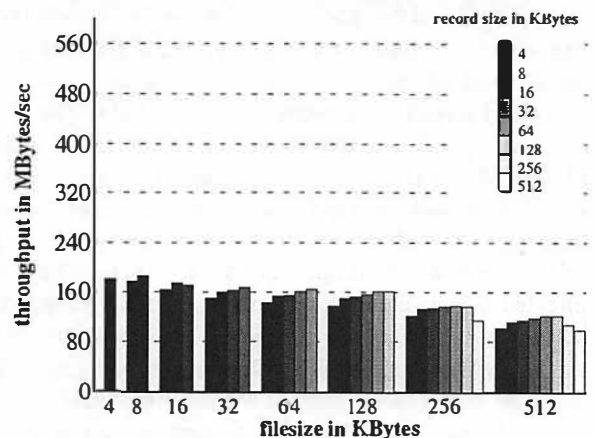
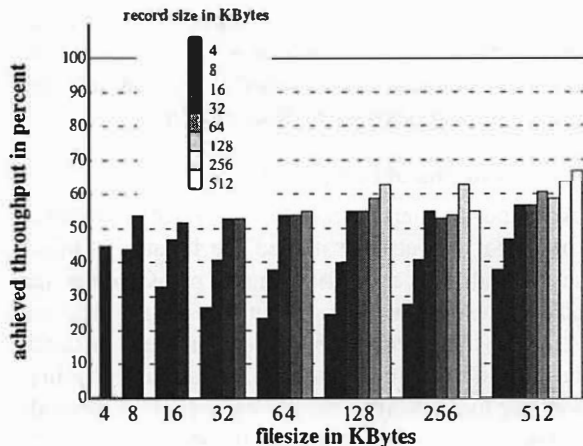
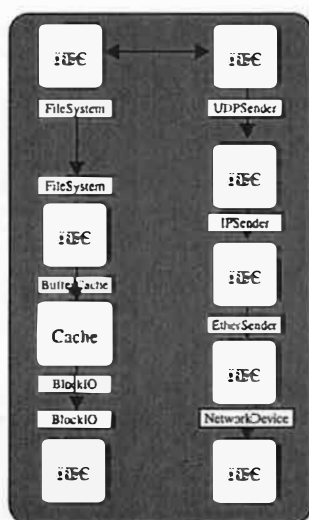


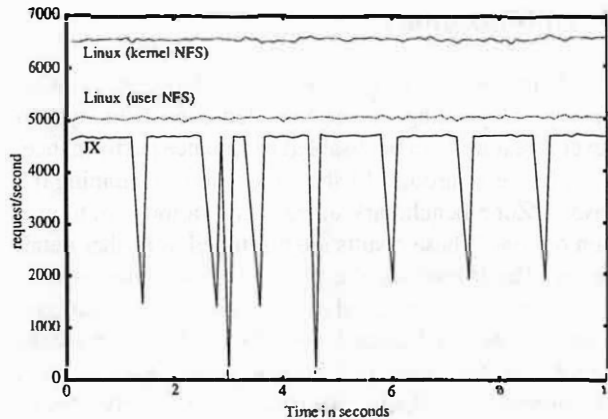
Figure 4: JX IOZone: multi-domain configuration



**Figure 5: JX vs. Linux: multi-domain configuration**  
for a dedicated NFS server. Figure 8 shows the results of running the rate benchmark with a Linux NFS server (both kernel and user-level NFS) and with a JX NFS server. There are drops in the JX request rate that occur very periodically. To see what is going on in the JX NFS server, we collected thread switch information and created a thread activity diagram. Figure 7 shows this diagram. We see an initialization phase which is completed six seconds after startup. Shortly after startup a periodic thread (ID 2.12) starts, which is the interrupt handler of the real-time clock. But the important activity starts at about 17 seconds. The CPU is switched between “IRQThread1”, “Etherpacket-Queue”, “NFSProc”, and “Idle” thread. This is the activity during the rate benchmark. Packets are received and put into a queue by the first-level interrupt handler of the network interface “IRQThread1” (ID 2.14). This unblocks the “Etherpacket-Queue” (ID 2.19), which processes the packet

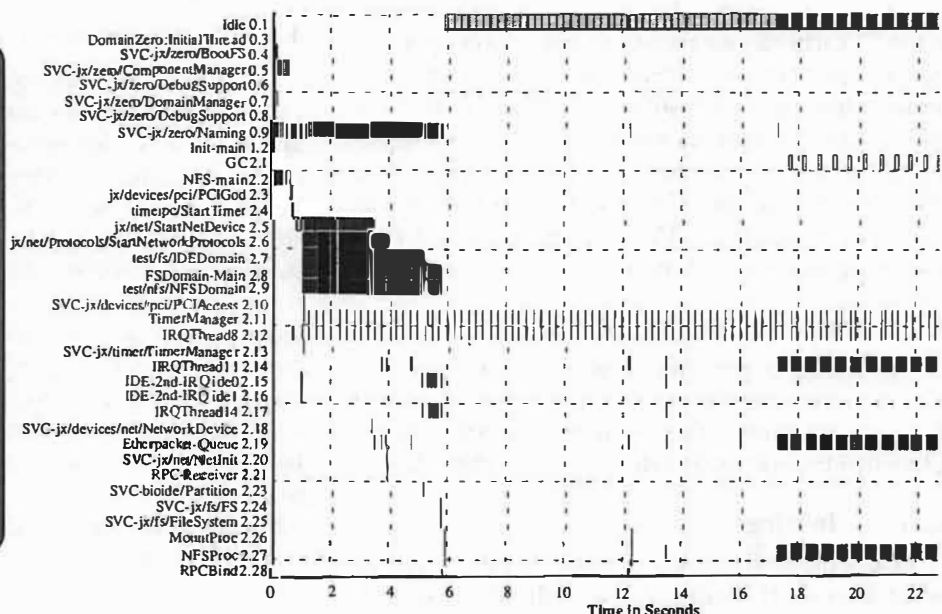


**Figure 6: JX NFS configuration**



**Figure 8: JX NFS performance (rate benchmark)**

and finally puts it into a UDP packet queue. This unblocks the “NFSProc” (ID 2.27) thread, which processes the NFS packet and accesses the file system. This is done in the same thread, because the NFS component and the file system are collocated. Then a reply is sent and all threads block, which wakes up the “Idle” thread (ID 0.1). The sharp drops in the request rate of the JX NFS server in Figure 8 correspond to the GC thread (ID 2.1) that runs for about 100 milliseconds without being interrupted. It runs that long because neither the garbage collector nor the NFS server are optimized. Especially the RPC layer creates many objects during RPC packet processing. The GC is not interrupted, because it disables interrupts as a safety precaution in the current implementation. The pauses could be avoided by using an incremental GC [6], which allows the GC thread to run concurrently with threads that modify the heap.



**Figure 7: Thread activity during the rate benchmark**

## 4 Optimizations

JX provides a wide range of flexible configuration options. Depending on the intended use of the system several features can be disabled to enhance performance.

Figures 9 through 14 show the results of running the Java IOZone benchmark on JX with various configuration options. These results are discussed in further detail below. The legend for the figures indicates the specific configuration options used in each case. The default configuration used in Figure 3 was MNNSCR, which means that the configuration options used were multi-domain, no inlining, no inlined memory access, safety checks enabled, memory revocation check by disabling interrupts, and a Java round-robin scheduler. At the end of this section we will select the fastest configuration and repeat the comparison to Linux.

The modifications described in this sections are pure configurations. Not a single line of code is modified.

### 4.1 Domain structure

How the system is structured into domains determines communication overheads and thus affects performance. For maximal performance, components should be placed in the same domain. This removes portal communication overhead. Figure 9 shows the improvement of placing all components into a single domain. The performance improvement is especially visible when using small record sizes, because then many invocations between the IOZone component and the filesystem component take place. The larger improvement in the 4KB file size / 4KB record size can be explained by the fact that the overhead of a portal call is relatively constant and the 4KB test is very fast, because it completely operates in the L1 cache. So the portal call time makes up a considerable part of the complete time. The contrary is true for large file sizes: the absolute throughput is lower due to processor cache misses and the saved time of the portal call is only a small fraction of the complete time. Within one file size the effect also becomes smaller with increasing record sizes. This can be explained by the decreasing number of performed portal calls.

### 4.2 Translator configuration

The translator performs several optimizations. This section investigates the performance impact of each of these optimizations. The optimizations are inlining, inlining of fast portals, and elimination of safety checks.

#### 4.2.1 Inlining

One of the most important optimizations in an object-oriented system is inlining. We currently inline only non-virtual methods (final, static, or private). We plan to

inline also virtual methods that are not overridden, but this would require a recompilation when, at a later time, a class that overrides the method is loaded into the domain. Figure 10 shows the effect of inlining.

#### 4.2.2 Inlining of fast portals

A fast portal interface (see Sec. 2.2) that is known to the translator can also be inlined. To be able to inline these methods that are written in C or assembler the translator must know their semantics. Since we did not want to wire these semantics too deep into the translator, we developed a plugin architecture. A translator plugin is responsible for translating the invocations of the methods of a specific fast portal interface. It can either generate special code or fall back to the invocation of the Domain-Zero method.

We did expect a considerable performance improvement but as can be seen in Figure 11 the difference is very small. We assume, that these are instruction cache effects: when a memory access is inlined the code is larger than the code that is generated for a function call. This is due to range checks and revocation checks that must be emitted in front of each memory access.

#### 4.2.3 Safety checks

Safety checks, such as stack size check and bounds checks for arrays and memory objects can be omitted on a per-domain basis. Translating a domain without checks is equivalent to the traditional OS approach of hoping that the kernel contains no bugs. The system is now as unsafe as a kernel that is written in C. Figure 12 shows that switching off safety checks can give a performance improvement of about 10 percent.

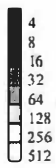
### 4.3 Memory revocation

Portals and memory objects are the only objects that can be shared between domains. They are capabilities and an important functionality of capabilities is revocation. Portal revocation is implemented by checking a flag before the portal method is invoked. This is an inexpensive operation compared to the whole portal invocation. Revocation of memory objects is more critical because the operations of memory objects - reading and writing the memory - are very fast and frequently used operations. The situation is even more involved, because the check of the revocation flag and the memory access have to be performed as an atomic operation. JX can be configured to use different implementations of this revocation check:

- **NoCheck:** No check at all, which means revocation is not supported.

## Legend for all figures on this page:

record size in KBytes



Encoding of the measured configuration:

1. domain structure: S (single domain), M (multi domain)
2. inlining: I (inlining), N (no inlining)
3. memory access: F (inlined memory access), N (no inlined memory access)
4. safety checks: S (safety checks enabled), N (safety checks disabled)
5. memory revocation: N (no memory revocation), C (disable interrupts), S (spinlock), A (atomic code)
6. scheduling: C (microkernel scheduler), R (Java RR scheduler), I (Java RR invisible portals)

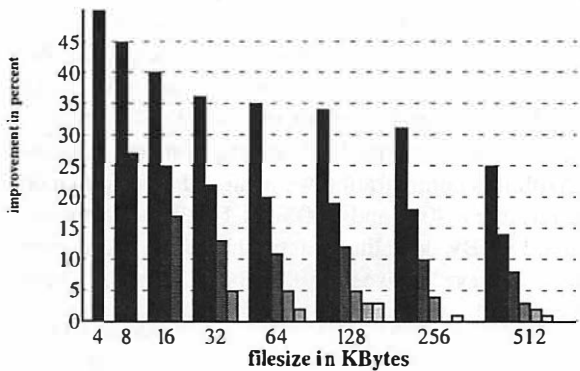


Figure 9: Domain structure: SNNSCR vs. MNNSCR

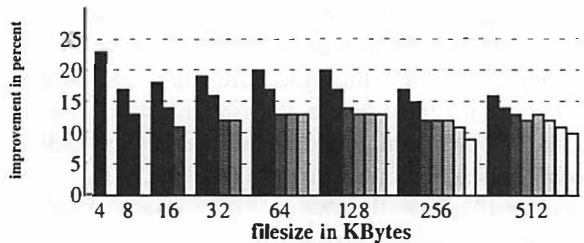


Figure 10: Inlining: SINSR vs. SNNSCR

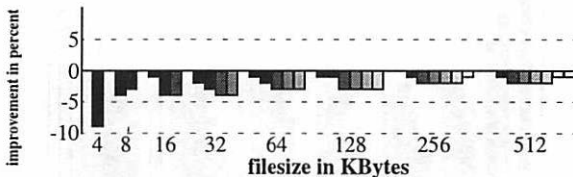


Figure 11: Memory access inlining: SIFSNR vs. SINSR

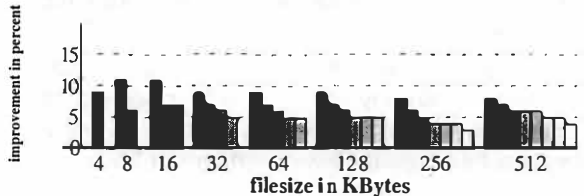


Figure 12: Safety checks: SIFNCR vs. SIFSCR

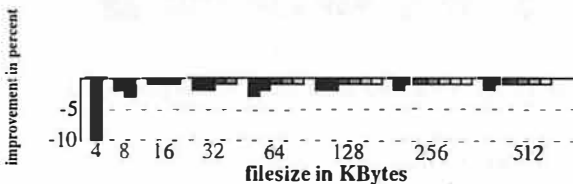


Figure 13a: No revocation: SIFSNR vs. SIFSCR

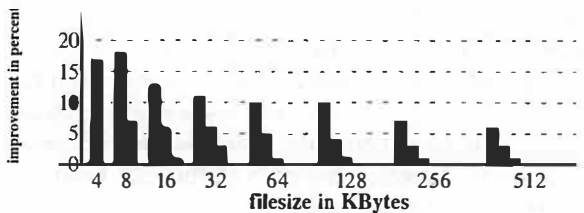


Figure 14a: Simple Java Scheduler: MIFSNR vs. MIFSNR

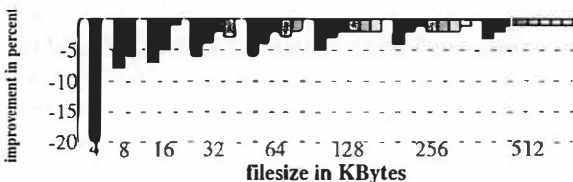


Figure 13b: SPIN revocation: SIFSSR vs. SIFSCR

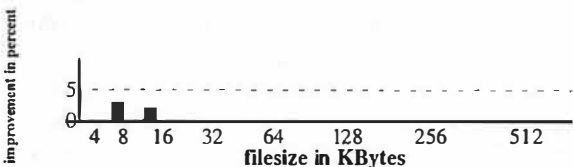


Figure 13c: ATOMIC revocation: SINSAR vs. SIFSCR

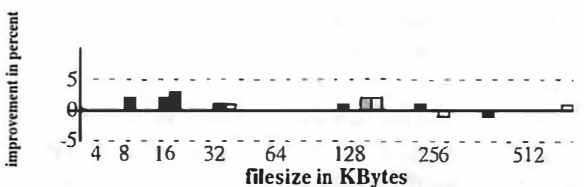


Figure 14b: Kernel scheduler: MIFSNC vs. MIFSNR

- **CLI:** Saves the interrupt-enable flag and disables interrupts before the memory access and restores the interrupt-enable flag afterwards.
- **SPIN:** In addition to disabling interrupts a spinlock is used to make the operation atomic on a multiprocessor.
- **ATOMIC:** The JX kernel contains a mechanism to avoid locking at all on a uniprocessor. The atomic code is placed in a dedicated memory area. When the low-level part of the interrupt system detects that an interrupt occurred inside this range the interrupted thread is advanced to the end of the atomic procedure. This technique is fast in the common case but incurs the overhead of an additional range check of the instruction pointer in the interrupt handler. It increases interrupt latency when the interrupt occurred inside the atomic procedure, because the procedure must first be finished. But the most severe downside of this technique is, that it inhibits inlining of memory accesses. Similar techniques are described in [9], [36], [35], [41].

Figure 13a shows the change in performance when no revocation checks are performed. This configuration is slightly slower than a configuration that used the CLI method for revocation check. We can only explain this by code cache effects.

Using spinlocks adds an additional overhead (Figure 13b). Despite some improvements in a former version of JX using atomic code could not improve the IOZone performance of the measured system (Figure 13c).

#### 4.4 Cost of the open scheduling framework

Scheduling in JX can be accomplished with user-defined schedulers (see Sec. 2.8). The communication between the global scheduler and the domain schedulers is based on interfaces. Each domain scheduler must implement a certain interface if it wants to be informed about special events. If a scheduler does not need all the provided information, it does not implement the corresponding interface. This reduces the number of events that must be delivered during a portal call from the microkernel to the Java scheduler.

In the configurations presented up to now we used a simple round-robin scheduler (RR) in each domain. The domain scheduler is informed about every event, regardless whether being interested in it or not. Figure 14a shows the benefit of using a scheduler which implements only the interfaces needed for the round-robin strategy (RR invisible portals) and is not informed when a thread switch occurred due to a portal call.

As already mentioned, there is a scheduler built into the microkernel. This scheduler is implemented in C and can not be exchanged at run time. Therefore this type of scheduling is mainly used during development or performance analysis. The advantage of this scheduler is that

there are no calls to the Java level necessary. Figure 14b shows that there is no relevant performance difference in IOZone performance between the core scheduler and the Java scheduler with invisible portals.

#### 4.5 Summary: Fastest safe configuration

After we explained all the optimizations we can now again compare the performance of JX with the Linux performance. The most important optimizations are the use of a single domain, inlining, and the use of the core scheduler or the Java scheduler with invisible portals. We configured the JX system to make revocation checks using CLI, use a single domain, use the kernel scheduler, enabled inlining, and disabled inlining of memory methods. With this configuration we achieved a performance between about 40% and 100% of Linux performance (Figure 15). By disabling safety checks we were even able to achieve between 50% and 120% of Linux performance.

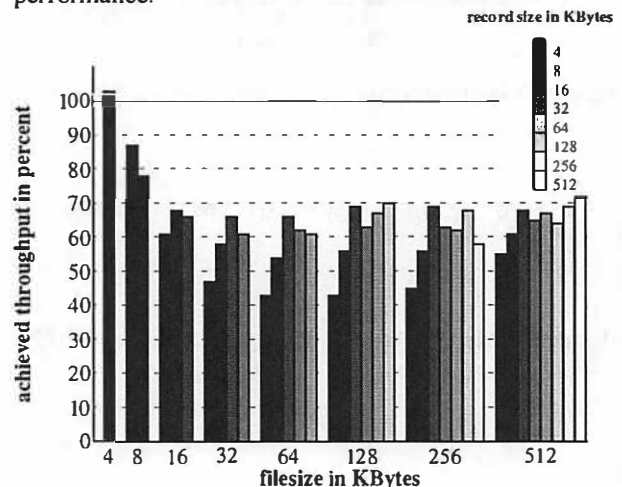


Figure 15: JX vs. Linux: Fastest configuration (SINSCC)

## 5 Related work

There are several areas of related work. The first two areas are concerned with general principals of structuring an operating system: extensibility and reusability across system configurations. The other areas are language-based operating systems and especially Java operating systems.

**Extensibility.** With respect to extensibility JX is similar to L4 [33], Pebble [25], and the Exokernel [24] in that it tries to reduce the fixed, static part of the kernel. It is different from systems like SPIN [8] and VINO [40], because these systems only allow a gradual modification of the system service, using spindles (SPIN) or grafts (VINO). JX allows its complete replacement. This is necessary in some cases and in most cases will give a better performance, because more suitable algorithms can



be used inside the service. A system service with an extension interface will only work as long as the extensions fit into a certain pattern that was envisioned by the designer of the interface. A more radical change of the service is not possible.

An important difference between JX and previous extensible systems is, that in JX the translator is part of the operating system. This allows several optimizations as described in the paper.

**Modularity and protection.** Orthogonality between modularity and protection was brought forward by Lipto [22]. The OSF [15] attacked the specific problem of collocating the OSF/1 UNIX server, which was run on top of the Mach microkernel, with the microkernel. They were able to achieve a performance only 8% slower than a monolithic UNIX. The special case of code reuse between the kernel and user environment was investigated in the Rialto system [21]. Rialto uses two interfaces, a very efficient one for collocated components (for example the mbuf [34] interface) and another one when a protection boundary must be crossed (the normal read/write interface). We think that this hinders reusability and complicates the implementation of components, especially as there exist techniques to build “unified” interfaces in MMU-based systems [23], and, using our memory objects, also in language-based systems.

There is a considerable amount of work in single address space operating systems, such as Opal [11] and Mungi [29]. Most of these systems use hardware protection, depend on the mechanisms that are provided by the hardware, and must structure the system accordingly, which makes their problems much different from ours.

**Language-based OS.** Using a safe language as a protection mechanism is an old idea. A famous early system was the Pilot [38], which used a language and bytecode instruction set called Mesa [30], an instruction set for a stack machine. Pilot was not designed as a multi-user operating system. More recent operating systems that use safe languages are SPIN [8], which uses Modula3, and Oberon [47], which uses the Oberon language, a descendant of Modula2.

**Java OS.** The first Java operating system was JavaOS from Sun [39]. We do not know any published performance data for JavaOS, but because it used an interpreter, we assume that it was rather slow. Furthermore, it did only provide a single protection domain. This makes sense, because JavaOS was planned to be a thin-client OS. However, besides JX, JavaOS is the only system that tried to implement the complete OS functionality in Java. JKernel [28], the MVM [16], and KaffeOS [4] are systems that allow isolated applications to run in a single JVM. These systems are no operating systems, but con-

tain several interesting ideas. JKernel is a pure Java program and uses the name spaces that are created by using different class loaders, as a means of isolation. JKernel concentrates on the several aspects how to implement a capability mechanism in pure Java. It relies on the JVM and OS for resource management. The MVM is an extension of Sun’s HotSpot JVM that allows running many Java applications in one JVM and give the applications the illusion of having a JVM of their own. It allows to share bytecode and JIT-compiled code between applications, thus reducing startup time. There are no means for resource control and no fast communication mechanisms for applications inside one MVM. KaffeOS is an extension of the Kaffe JVM. KaffeOS uses a process abstraction that is similar to UNIX, with kernel-mode code and user-mode code, whereas JX is more structured like a multi-server microkernel system. Communication between processes in KaffeOS is done using a shared heap. Our goal was to avoid sharing between domains as much as possible and we, therefore, use RPC for inter-domain communication. Furthermore, KaffeOS is based on the Kaffe JVM, which limits the overall performance and the amount of performance optimizations that are possible in a custom-build translator like ours.

These three systems do not have the robustness advantages of a 100% Java OS, because they rely on a traditional OS which is written in a low-level language, usually C.

## 6 Conclusion and future work

We described the JX operating system and its performance. While being able to reach a performance of about 50% to 100% of Linux in a file system benchmark in a monolithic configuration, the system can be used in a more flexible configuration with a slight performance degradation.

To deliver our promise of outperforming traditional, UNIX-based operating systems, we have to further improve the translator. The register allocation is still very simple, which is especially unsatisfactory on a processor with few registers, like the x86.

We plan to refine the memory objects. Several additional memory semantics are possible. Examples are copy-on-write memory, a memory object that represents non-continuous chunks of memory as one memory object, or a memory object that does not allow revocation. All these semantics can be implemented very efficiently using compiler plugins. The current implementation does not use an MMU because it does not need one. MMU support can be added to the system to expand the address space or implement a copy-on-write memory. How this complicates the architecture and its implementation remains to be seen.



## 7 Acknowledgements

We wish to thank the anonymous reviewers and our shepherd Jason Nieh for the many comments that helped to improve the paper. Franz Hauck and Frank Bellosa read an earlier version of the paper and suggested many improvements.

## 8 References

- [1] B. Alpern, A. Cocchi, S. J. Fink, D. P. Grove, and D. Lieber. Efficient Implementation of Java Interfaces: invokeinterface Considered Harmless. In *OOPSLA 01*, Oct. 2001.
- [2] M. Alt. Ein Bytecode-Verifizierer zur Verifikation von Betriebssystemkomponenten. Diplomarbeit, available as DA-14-2001-10, Univ. of Erlangen, Dept. of Comp. Science, Lehrstuhl 4, July 2001.
- [3] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM Trans. on Computer Systems*, 10(1), pp. 53-79, Feb. 1992.
- [4] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proc. of 4th Symposium on Operating Systems Design & Implementation*, Oct. 2000.
- [5] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the Design of Java Operating Systems. In *2000 USENIX Annual Technical Conference*, June 2000.
- [6] H. G. Baker. List processing in real time on a serial computer. In *Communications of the ACM*, 21(4), pp. 280-294, Apr. 1978.
- [7] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. In *Operating Systems Review*, 23(5), pp. 102-113, Dec. 1989.
- [8] B. Bershad, S. Savage, P. Pardyak, E. G. Siter, D. Becker, M. Fluczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. of the 15th Symposium on Operating System Principles*, pp. 267-284, Dec. 1995.
- [9] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 223-233, Sep. 1992.
- [10] R. Campbell, N. Islam, D. Raila, and P. Madany. Designing and Implementing Choices: An Object-Oriented System in C++. In *Communications of the ACM*, 36(9), pp. 117-126, Sep. 1993.
- [11] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single Address Space Operating System. In *ACM Trans. on Computer Systems*, 12(4), pp. 271-307, Nov. 1994.
- [12] T. M. Chilimbi. Cache-Conscious Data Structures - Design and Implementation. Ph.D. thesis, University of Wisconsin-Madison, 1999.
- [13] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Symposium on Operating System Principles 01*, 2001.
- [14] R. P. Colwell, E. F. Gehring, and E. D. Jensen. Performance effects of architectural complexity in the intel 432. In *ACM Trans. on Computer Systems*, 6(3), pp. 296-339, Aug. 1988.
- [15] M. Condit, D. Bolinger, E. McManus, D. Mitchell, and S. Lewontin. *Microkernel modularity with integrated kernel performance*. Technical Report, OSF Research Institute, Cambridge, MA, Apr. 1994.
- [16] G. Czajkowski and L. Daynes. Multitasking without Compromise: A Virtual Machine Evolution. In *Proc. of the OOPSLA*, pp. 125-138, Oct. 2001.
- [17] P. Dasgupta, R. J. LeBlanc, M. Ahmad, and U. Ramachandran. The Clouds distributed operating system. In *IEEE Computer*, 24(11), pp. 34-44, Nov. 1991.
- [18] J. B. Dennis and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. In *Communications of the ACM*, 9(3), pp. 143-155, Mar. 1966.
- [19] Department of Defense. *Trusted computer system evaluation criteria (Orange Book)*. DOD 5200.28-SID, Dec. 1985.
- [20] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 211-214, 1989.
- [21] R. Draves and S. Cutshall. *Unifying the User and Kernel Environments*. Technical Report MSR-TR-97-10, Microsoft Research, Mar. 1997.
- [22] P. Druschel, L. L. Peterson, and N. C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proc. of Twelfth International Conference on Distributed Computing Systems*, pp. 512-520, 1992.
- [23] P. Druschel and L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *14th ACM Symp. on Operating System Principles*, pp. 189-202, 1993.
- [24] D. Engler, F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th Symposium on Operating System Principles*, pp. 251-266, Dec. 1995.
- [25] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-Based Operating System. In *USENIX 1999 Annual Technical Conference*, pp. 267-282, June 1999.
- [26] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Aug. 1996.
- [27] G. Hamilton and P. Kougioris. The Spring Nucleus: a Micro-kernel for objects. In *Proc. of Usenix Summer Conference*, pp. 147-159, June 1994.
- [28] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. v. Eicken. Implementing Multiple Protection Domains in Java. In *Proc. of the USENIX Annual Technical Conference*, pp. 259-270, June 1998.
- [29] G. Heiser, K. Elphinstone, J. Vochtelloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. In *Software: Practice and Experience*, 28(9), pp. 901-928, Aug. 1998.
- [30] R. K. Johnson and J. D. Wick. An overview of the Mesa processor architecture. In *ACM Sigplan Notices*, 7(4), pp. 20-29, Apr. 1982.
- [31] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun Unix. In *USENIX Association: Summer Conference Proceedings*, 1986.
- [32] I4Ka Hazelnut evaluation, <http://i4ka.org/projects/hazelnut/eval.asp>.
- [33] J. Liedtke. Towards Real u-Kernels. In *CACM*, 39(9), 1996.
- [34] M. K. McKusick, K. Bostic, and M. J. Karels. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, May 1996.
- [35] M. Michael and M. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. In *Journal of Parallel and Distributed Computing*, 54(2), pp. 162-182, 1998.
- [36] D. Mosberger, P. Druschel, and L. L. Peterson. Implementing Atomic Sequences on Uniprocessors Using Rollforward. In *Software—Practice and Experience*, 26(1), pp. 1-23, Jan. 1996.
- [37] E. I. Organick. *A Programmer's View of the Intel 432 System*. McGraw-Hill, 1983.
- [38] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: An operating system for a personal computer. In *Communications of the ACM*, 23(2), pp. 81-92, ACM Press, New York, NY, USA, Feb. 1980.
- [39] T. Saulpaugh and C. Mirho. *Inside the JavaOS Operating System*. Addison Wesley Longman, 1999.
- [40] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *2nd Symposium on Operating Systems Design and Implementation*, 1996.
- [41] O. Shivers, James W. Clark, and Roland McGrath. Atomic heap transactions and fine-grain interrupts. In *ACM Sigplan International Conference on Functional Programming (ICFP99)*, Sep. 1999.
- [42] Status page of the Fiasco project at the Technical University of Dresden, <http://os.inf.tu-dresden.de/fiasco/status.html>.
- [43] Sun Microsystems. *Java Remote Method Invocation Specification*. 1997.
- [44] Webpage of the IOZone filesystem benchmark, <http://www.iozone.org/>.
- [45] A. Weissel. *Ein offenes Dateisystem mit Festplattensteuerung fuer metaXaOS*. Studienarbeit, available as SA-14-2000-02, Univ. of Erlangen, Dept. of Comp. Science, Lehrstuhl 4, Feb. 2000.
- [46] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proc. of International Workshop on Memory Management*, Sep. 1995.
- [47] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992.
- [48] F. Yellin. Low level security in Java. In *Proc. of the 4th World Wide Web Conference*, pp. 369-379, O'Reilly, 1995.

# Design Evolution of the EROS Single-Level Store\*

Jonathan S. Shapiro  
Systems Research Laboratory  
Johns Hopkins University  
shap@cs.jhu.edu

Jonathan Adams  
Distributed Systems Laboratory  
University of Pennsylvania†  
jonathan-adams@ofb.net

## Abstract

File systems have (at least) two undesirable characteristics: both the addressing model and the consistency semantics differ from those of memory, leading to a change in programming model at the storage boundary. Main memory is a single flat space of pages with a simple durability (persistence) model: all or nothing. File content durability is a complex function of implementation, caching, and timing. Memory is globally consistent. File systems offer no global consistency model. Following a crash recovery, individual files may be lost or damaged, or may be *collectively* inconsistent even though they are individually sound.

Single level stores offer an alternative approach in which the memory system is extended all the way down to the disk level. This extension is accompanied by a transacted update mechanism that ensures globally consistent durability. While single level stores are both simpler and potentially more efficient than a file system based design, relatively little has appeared about them in the public literature. This paper describes the evolution of the EROS single level store design across three generations. Two of these have been used; the third is partially implemented. We identify the critical design requirements for a successful single level store, the complications that have arisen in each design, and the resolution of these complications.

As the performance of the EROS system has been discussed elsewhere, this paper focuses exclusively on *design*. Our objective is to both clearly express how a single level store works and to expose some non-obvious details in these designs that have proven to be important in practice.

## 1 Introduction

Single level stores simplify operating system design by removing an unnecessary layer of abstraction from the system. Instead of implementing a new and different semantics at the file system layer, a single level store extends the memory mapping model downwards to include the disk. Where conventional operating systems use the memory mapping hardware to translate virtual page addresses to physical pages, single level stores map virtual page addresses to *logical* page addresses, using physical memory as a software-managed cache to hold these pages.

The most widely-used single level store design is probably the IBM System/38, more commonly known as the AS/400 [IBM98]. At the hardware level, the AS/400 is a capability-based object system. The AS/400 design treats the entire store as a unified, 64-bit address space. Every object is assigned a 16 megabyte segment within this space. Persistence is managed explicitly – changes to objects are rewritten to disk only when directed by the application. While the protection architecture and object structure of the AS/400 is described by Soltis [Sol96], key details of its single-level store implementation are unpublished.

Like the AS/400, EROS is a capability-based single level

store design. Unlike AS/400, EROS manages persistence transparently using an efficient, transacted checkpoint system that runs periodically in the background. Applications rely on the kernel to transparently handle persistence, leaving the applications free to build data structures and algorithms without regard to disk-level placement or the need to protect recoverability through careful disk write ordering. The EROS system and its performance have been described elsewhere [SSF99]. This paper describes how the EROS single level store design is integrated into the system and its key components, and the evolution of this design over the last decade.

As an initial intuition for single level stores, imagine a system design that begins by assuming that the machine never crashes. In such a system, there would be no need for a file system at all; the entirety of the disk is used as a large paging area. Such a design would clearly eliminate a large body of code from a conventional operating system. The EROS system, including user-mode applications that implement essential functions, is currently 103,712 lines of code.<sup>1</sup> Excluding drivers, networking protocols, and include files, the Linux 2.4 kernel contains 383,698 lines of code, of which 283,956 implement support for various file systems and file mapping. While the two systems implement very different semantics, their functionality is comparable, and the EROS code provides features

\* This research was supported by DARPA under contract #N66001-96-C-852. Additional support was provided by Panasonic Information Technology Laboratories, Inc. and VMware, Inc.

† Author now with Sun Microsystems, Inc.

<sup>1</sup> EROS drivers and network stack are implemented outside the kernel. Driver and network code size is not included in either estimation of code size.

that Linux lacks: on restart, EROS recovers processes and interprocess communication channels in addition to data objects.

The design challenges of a single-level store are (1) to address the problem that systems actually *do* crash, and ensure that consistency is preserved when this occurs, (2) to devise some efficient means of addressing this very large paging area, and (3) to provide some means for specifying desired locality, preferably in a fashion informed by application-level semantic knowledge. This paper describes three designs that meet these challenges in two different EROS kernel designs. Other potential applications of these design ideas include database storage managers, storage-attached networks, and logical volume systems.

The first design presented is the one used by KeyKOS, which was inherited by the original EROS system in 1991. This design suffered from minor irritations that caused us to revise the design in 1997. In 2001, it was decided to remove drivers from the EROS kernel entirely, which forced us to rethink and partially rebuild the single level store yet again. Aside from the storage allocator itself, all of these designs present identical external semantics to applications.

The balance of this paper proceeds as follows. We first provide a brief overview of the EROS object system, its storage model and the mechanism used to ensure global consistency. This discussion introduces the critical requirements that must be satisfied by a single-level store design. We then describe the user-level storage allocator, which bears responsibility for locality management and storage reclamation. We then describe each of the existing design generations in turn, and the motivation behind each revision. The paper concludes with related work, lessons learned, and some hints as to our future plans.

## 2 Object System Overview

EROS is a microkernel design. The kernel implements a small number of object types, but leaves storage allocation, fault handling, address space management, and many other traditional kernel functions to user-level code. For this paper, the most important function implemented by user-level code is the *space bank* (Section 5). The space bank is responsible for all storage allocation, for storage quota enforcement, for bulk storage reclamation, and for disk-level object placement. At the kernel interface, all of this is accomplished by allocating and deallocating objects with appropriately selected unique object identifiers. Every object has a unique object identifier (OID). OIDs directly correlate to disk locations, which enables the space bank to perform object placement.

The EROS kernel design consists of three layers (Fig-

ure 1). The *machine layer* stores process and memory mapping information using a representation that is convenient to the hardware. For process state, this representation is determined by the design of the hardware context switch mechanism. For memory mapping state, it is determined by the design of the hardware memory management unit. These layers are managed as a cache of selected objects that logically reside in the object cache. When necessary, entries in the process cache or memory mapping tables are either written back or invalidated. Entries in the process cache correspond to entries in the process table of a more conventional design, but processes may be moved in and out of the process cache several times during their lifespan.

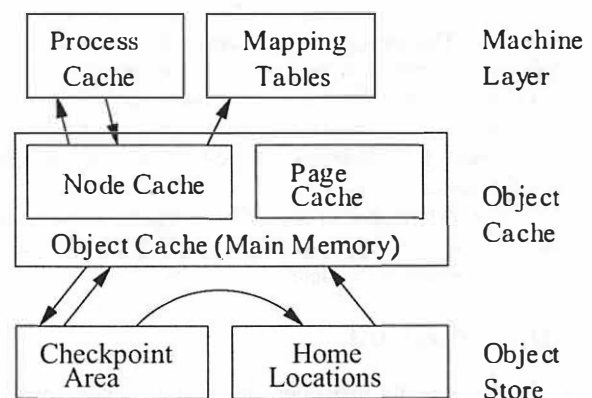


Figure 1: EROS design layers.

The *object cache* occupies the bulk of main memory. At this layer there are only two types of objects: pages and nodes. Pages hold user data. Nodes hold capabilities. Every node and page has a corresponding location in the home location portion of the object store. As with the machine layer, the object cache is a software-managed cache of the state on the disk.

As is implied by Figure 1, all higher-level operating system abstractions are composed from these two fundamental units of storage. Process state is stored in nodes, and is loaded into the process cache at need. Address space mappings are likewise represented using trees of nodes. These are traversed to construct hardware mapping data structures as memory faults occur. Details of these transformations can be found in [SSF99, SFS96].

The *object store* layer is the object system as it exists on the disk. At this layer the system is divided into two parts: the “home locations,” which provide space for every object in the system, and the “checkpoint area,” which provides the means for building consistent snapshots of the system. All object writes are performed to the checkpoint area. Revised objects are migrated to their home locations only after a complete, system-wide transaction has

been successfully committed. The check point mechanism is described in Section 3.2.

Collectively, these layers implement a two-level caching design. At need, the entire user-visible state of the system can be reduced to pages and nodes, which can then be written to disk.

### 3 Storage Model

The main reason for having two types and sizes of objects is to preserve a partition between data and capabilities. Data resides in pages and capabilities reside in nodes. It is certainly possible to design a single level store in which all objects are the size of a page, but it proved inconvenient to do so in EROS for reasons of storage efficiency. In the current implementation, capabilities occupy 16 bytes and nodes 544 bytes, but these sizes are not exposed by any external system interface. This leaves us free in the future to change the size of capabilities compatibly, much as was done in the AS/400 transition from 48-bit to 64-bit addresses.

Every object in the store has a unique object identifier (OID). Objects on the disk are named by operating-system protected capabilities [Dv66], each of which contains an object type (node, page, or process), the OID of the object it names, and a set of permissions. An EROS object capability is similar to a page table entry (PTE) that contains the swap location of an out-of-memory page.<sup>2</sup> When an object capability is actively in use, the EROS kernel rewrites the capability internally to point directly at the in-memory copy of the object.

Pages contain user data. Nodes contain a fixed-size array of capabilities, and are used as indirect blocks, memory mapping tables, and as the underlying representation for process state. Within the store, these nodes are packed into page-sized containers called “node pots.” All I/O to or from the store is performed in page-sized units.

#### 3.1 Home Locations

Every object in the EROS store has a uniquely assigned “home location” on some disk. Some versions of EROS implement optional object mirroring, in which case the same object may appear on the disk at multiple locations and is updated (when needed) at all locations. Additional mirroring or RAID storage may be performed by the storage controller. This is invisible to the EROS kernel.

The bulk of the disk space in an EROS system is used to contain the “home locations” of the objects. The basic design requirements for this part of the store are:

- Object fetch and store should be efficient. As a result, there should be a simple, in-memory strategy for directly translating an OID value to the disk page frame (the home location) that contains the object.
- EROS does not expose the physical locations of objects outside the kernel; only OIDs are visible, and these only to selected applications. For purposes of locality management, there must be some well-defined relationship between OID values and disk locations.

Without an in-memory algorithm to translate an OID into a disk object address, the store would require disk-level directory data structures that would in turn require additional, sequentially dependent I/O accesses to locate and fetch an object. The elimination of such additional accesses is a performance-critical imperative. The encoding of object locations, the organization of the disk, and the locality management of home locations has been the main focus of evolution in the design of the storage manager.

#### 3.2 Checkpointing

To ensure that global consistency for all processes and objects is maintained across restarts, it is sufficient for the kernel to periodically write down an instantaneous snapshot of the state of the corresponding pages and nodes. To accomplish this, EROS implements an efficient, asynchronous, system-wide check point mechanism derived from the check point design of KeyKOS [Lan92].

The checkpoint system makes use of a dedicated area on the disk. This area is also used for normal paging, and is conceptually equivalent to the “swap area” of a conventional paging system. Before any node or page is made dirty, space is reserved for it in the checkpoint area. As memory pressure induces paging, dirty objects are paged out to (and if necessary, reread from) the checkpoint area. Object locations in the checkpoint area are recorded in an in-memory directory.

Periodically, or when the checkpoint area has reached a predefined occupancy threshold, the kernel declares a “snapshot,” in which every dirty object in memory is marked “copy on write.” Simultaneously, a watermark is made in the checkpoint area. Everything modified prior to the snapshot will be written beneath this watermark; everything modified after the snapshot is written above it. The kernel now allows execution to proceed, and initiates background processing to flush all of the pre-snapshot dirty objects into the previously reserved space in the checkpoint area. Checkpoint area I/O is append-only. If an object is dirtied multiple times, no attempt is made to reclaim its

<sup>2</sup> In the PTE case, no object type is needed because PTEs can only name pages.

previously occupied space in the checkpoint area. This ensures that checkpoint I/O is mostly sequential.

Once all objects have been written to the checkpoint area, an area directory is written and a log header is rewritten to capture the fact that a consistent system-wide transaction has been completed. A background “migrator” now copies these objects back to their home locations in the store. Whenever a checkpoint transaction completes, the checkpoint area space occupied by the previous transaction is released. To ensure that there is always space in the checkpoint area for the next checkpoint, migration is required to complete before a new checkpoint transaction can be completed.

The net effect of the checkpoint system is to capture a consistent system-wide image of the system state. If desired, checkpoints can be run at frequencies comparable to those of conventional buffer cache flushes, making the potential loss of data identical to that of conventional systems. To support the requirements of database logs, there is a special “escape hatch” mechanism permitting immediate transaction of individual pages.

### 3.3 Intuitions for Latency

While EROS is not yet running application code, KeyKOS has been doing so since 1980, supporting both transaction processing and (briefly) general purpose workloads. The performance of the checkpoint design rests on two empirical observations from KeyKOS:

- Over 85% of disk reads are satisfied from the checkpoint area.
- Over 50% of dirty objects die or are redirtied before they are migrated. Such objects do not require migration.

These two facts alter the seek profile of the system, reducing effective seek latencies for reads. They also alter the rotational delay profile of the system, reducing effective rotational latencies for writes. Our goal in this section is to provide an intuition for why this is true. While the specific measurements obtained from KeyKOS probably will not hold for EROS twenty years later, we expect that the performance of checkpointing will remain robust. We will discuss the reasons for this expectation below.

It is typical for the amount of data included in a given checkpoint to be comparable to the size of main memory. The checkpoint area as a whole must be able to hold two checkpoints. Given a machine with 256 megabytes of memory, the expected checkpoint area would be 512 megabytes. On a Seagate Cheetah (ST373405LC, 29550 cylinders, 68 Gbytes), this region would occupy 0.7% of the disk, or 216 cylinders. As 100% of normal writes and

85% of all reads occur within this region, the arm position remains within a very narrow range of the disk with high probability.

Estimating disk latencies is deceptive, because computations based on the published minimum, average, maximum seek times have nothing to do with actual behavior. Seek time profiling is required for effective estimation. The seek time calculations presented here are based on profiling data collected by Jiri Schindler using DIXtrac [SG99]. We emphasize that these are computed, rather than measured results. We will assume a disk layout in which the checkpoint area is placed on the middle cylinders of the drive.

#### 3.3.1 Expected Read Behavior

The disk head position in KeyKOS has a non-uniform distribution. To compute the expected seek time, we must consider the likely location of the preceding read as well as the current one (Table 1), giving an expected seek time for reads of 2.92ms on a drive whose *average* seek time is 6.94 ms. This expectation is robust in the face of both changes to the checkpoint region size and reasonable reductions in checkpoint locality. Increasing the checkpoint area size to 200 cylinders raises the expected seek time to 3.05 ms. Reducing the checkpoint “hit rate” to 70% yields an expected seek time of 3.80 ms.

Conventional file system read performance is largely determined by the average seek delay (in this case, 6.94 ms). For comparing read delays, rotational latencies can be ignored: the expected rotation delay on both systems is one half of a rotation per read.

The difference in expected performance is largely immune to changes in extent size or prefetching, as both techniques can be used equally well on both systems. Both techniques reduce the total number of seeks performed; neither alters the underlying seek latency or distribution. Similarly, low utilization yields similar benefits in both systems by reducing the effect of long seeks. The read performance of both designs converges on the performance of the checkpointing design as utilization falls – on sufficiently small, packed data sets there is no meaningful difference in seek behavior.

#### 3.3.2 Expected Write Behavior

As in log-based file systems, a checkpointing design potentially performs two writes for every dirty block: one to the checkpoint area and the second to the home locations. Migration is skipped for data that is remodified or deleted between the time of checkpoint and the time of migration. Given this, there are two thresholds of interest:



Current I/O	Preceding I/O	Distance	Time	Weighted
Checkpoint (85%)	Checkpoint (85%)	108 cyl	1.97 ms	1.42 ms
Checkpoint (85%)	Other (15%)	7387 cyl	5.27 ms	0.67 ms
Other (15%)	Checkpoint (85%)	7387 cyl	5.27 ms	0.67 ms
Other (15%)	Other (15%)	14775 cyl	<b>6.94 ms</b>	0.16 ms
<b>Weighted Seek Time</b>				2.92 ms

Table 1: **Expected read latency.** Based on seek profile of the Seagate Cheetah ST373405Lc. The reported average seek time for this drive is 6.94 ms.

1. How many objects live long enough to get checkpointed.
2. Of those, how many live long enough to be migrated.

The best available data on file longevity is probably the data collected by Baker *et al.* [BHK<sup>+</sup>91]. Figure 4 of their paper indicates that 65% to 80% of all files live less than 30 seconds, that 80% of all files live less than 300 seconds, and that 90% of all files live less than 600 seconds (one checkpoint interval). We can estimate from this that less than half of all files that are checkpointed survive to be migrated. This is consistent with the measured behavior of KeyKOS: only 50% of the checkpoint data survives to be migrated.

The impact of this is surprising. Imagine that there are 400 kilobytes of file data to be written to a conventional file system. The key question proves to be: what are the run lengths? Figure 1 of the Baker measurements [BHK<sup>+</sup>91] shows that most file run lengths are small. While the figure does not differentiate read and write run lengths, Table 3 suggests that write run lengths are primarily driven by file size: 70% of all bytes written are “whole file” writes. Figure 2 shows that 80% of files are 10 kilobytes or less. Taken together, these numbers mean that the cost of bulk flushes of the data cache are dominated by rotational delay. The 400 kilobytes in question will be written at nearly 40 distinct locations, each of which will require 1/2 a rotation to bring the head to the correct position within the track. On the Cheetah, the rotational delay alone comes to 119 ms. Seek delays depend heavily on filesystem layout, but the same considerations apply in both checkpointed and conventional designs. If the runs are uniformly spread across the drive, the seeks (on the Cheetah) will come to an additional 112 ms, for a total of **231 ms**.

Now consider the same 400k under the checkpointing design. KeyKOS and EROS perform this write using bulk I/O and track at once operations. Depending on the drive, 400 to 600 kilobytes can be written to the checkpoint area in one seek (weighted cost 2.46ms on the Cheetah) plus 1.5 rotations (1/2 to start, 1 to complete) for a total of 11.42 ms. We must now consider the cost of migration.

The KeyKOS/EROS migration I/O behavior looks exactly like that of a file cache that uses deferred writes. Because the file semantics is unchanged this I/O has similar run lengths, and like the buffer cache flush it is done using bulk-sorted I/O. Seek times are amortized similarly because there are a large number of available blocks to write. The difference is that the migrated blocks have a longer time to die, and that the amount of data migrated is therefore half of the data that will be written by the deferred-write buffer cache. Because half of the data will die before migration, only 56 ms will be spent in rotational delay rewriting it and 70.47 ms of seek times (again under the uniform distribution assumption). The combined total cost of the checkpoint and migration writes is **137.89 ms**.

## 4 Locality and Object Allocation

There are two primary issues that impact the design of a single level store. The first is common to all disk-based storage designs: locality. It is necessary that the object allocation mechanism provide means to arrange the disk-level placement of objects for reasonable locality. In all generations of the EROS store this is accomplished by preserving a correlation between OID values and disk positions. The second is object allocation: because all objects must be recoverable after a crash, all allocations must (logically) be recorded using on-disk data structures.

### 4.1 Content Locality

The value of locality in general-purpose workloads is often misunderstood. While sequential data placement for file content is extremely important in the case of a single request stream, it is much less important when multiple accesses to disk occur concurrently. Disk-level traces collected by Ruemmler and Wilkes [RW93] show that it is very rare to see more than 8 kilobytes of sequential I/O at the level of the disk arm. While average file sizes have grown since that time, and (we presume) modern sequential accesses would be longer than 8 kilobytes, the underlying reasons for the limited dynamic sequentiality have not changed:



- Paging I/O is limited by the size of a page.
- Many file I/Os involve sequentially dependent accesses, as when traversing metadata.
- Directory I/Os are frequent. Directories are usually small, and the corresponding I/Os are therefore short.
- While read-ahead helps, excessive read ahead is counterproductive. Successful read-ahead works equally well in both designs, and can be thought of as achieving a larger extent size.
- The request streams compete for attention at the disk arm. Even if a second, potentially sequential I/O request is initiated quickly by the application or the operating system, the interrupt-level logic has already initiated an arm motion if multiple requests are present, preventing immediate service of the sequential request.

Two facts suggest that file system sequentiality may be important only up to a limited extent size. Log-structured designs organize data by temporal locality rather than spatial locality. In spite of this, read performance for general-purpose workloads is not degraded significantly in log-structured designs [SSB<sup>+</sup>95]. It has also been established that file system aging ultimately has a more significant impact on overall I/O performance than the logging/clustering choice [SS95].

The EROS store design effectively optimizes for both cases. Newly modified objects are stored in the checkpoint area according to temporal locality, but the small size of the checkpoint area ensures physical locality as well. Data in home locations is placed according to locality determined at object allocation time, which is when maximal semantic knowledge (and therefore maximal knowledge of likely reference patterns) is available.

While single level stores do not always implement directories and indirect blocks in the style of file systems, the corresponding *concepts* are implemented elsewhere in the operating system, and the basic usage patterns involved are ultimately driven by application behavior. Similar extent size arguments should therefore apply in single level stores. This introduces a significant degree of freedom into file system or single level store design — one that we plan to leverage in the next generation of our store.

## 4.2 Metadata Locality

A more pressing issue in the EROS single-level store is *metadata* locality. In the Berkeley Fast File System design [MJLF84], for example, block location occurs through a

two-stage hybrid translation scheme. The first stage translates the inode number to the inode data structure. This translation is performed at file open time, and the result is cached in an in-memory inode. The second stage traverses the file indirect blocks to locate individual blocks in the file. These indirect blocks are cached in memory according to the same rules as other data blocks, but due to higher frequency of access are likely to remain in memory for active files.

An EROS address space is a persistent mapping from offsets to bytes. Address spaces need not be associated with processes, and EROS therefore uses them to hold file data as well as application memory images. As a result, the EROS metadata that is most closely analogous to conventional file metadata is EROS address space metadata.

An EROS address space is organized as a tree of nodes whose leaves are pages (Figure 2), much as a UNIX file is organized as a tree of indirect blocks whose leaves are data blocks. Because EROS nodes are much “narrower” than typical indirect blocks, the height of the address space tree for any given address space is taller than the height of the indirect block tree for a UNIX file of corresponding file ( $\log_{32}(\text{size}) > \log_{256}(\text{size})$ ). Any tree traversal of this type implies sequential disk accesses with associated seek delays. The Ruemmler data shows that these seeks are frequently interspersed with other accesses when multiple request streams are present.

Request interleave in turn interacts badly with typical, non-backtracking disk arm scheduling policies, and can lead to as much as a full disk seek before the next block down the tree will be fetched.<sup>3</sup> Because of their greater tree height, EROS address spaces potentially involve more levels of traversal, and it is correspondingly more important to managing the locality and prefetching of nodes within an address space. The discussion of the EROS space bank below (Section 5) describes how this locality is achieved.

## 4.3 Allocation Performance

The final performance issue in single level stores is the efficiency of object allocation — particularly with respect to ephemeral allocations such as heap pages or short-lived file content. In a conventional file system, these ephemeral blocks are allocated from swap space and do not survive system shutdown or failure. Because there is no expectation that these allocations are preserved across restarts, in-memory data structures and algorithms can be used to implement them. Linux, for example, keeps an in-memory allocation bitmap for each swap area [BC00].

There is no way to persistently store ephemeral allocation

<sup>3</sup> Many newer drives implement backtracking seeks, but doing so raises both convergence and variance issues that must be avoided by the operating system in real-time applications.

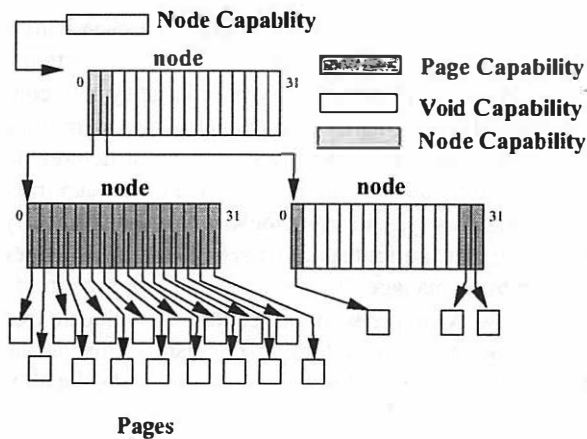


Figure 2: An EROS address space.

data without incurring *some* disk I/O overhead. The challenge in a single-level store is to keep this overhead to a minimum. EROS accomplishes this in two ways:

1. No recording of allocation is performed for objects that are allocated and deallocated within the same checkpoint interval. This largely recaptures the efficiency advantages of conventional swap area allocations.
2. In the current and previous versions of the EROS store, the store is divided into regions, each of which has an overhead page containing bits that indicate whether an object in the region is empty (zero). Sequential allocations first pull in the overhead page, but then avoid I/O's for successive objects within the same region. Deallocations update the bit rather than the on-disk object, with similar I/O reductions.

## 5 The Space Bank

The EROS storage manager, known as the “space bank,” is a user-mode application that performs all storage allocation in the EROS system. There is a hierarchy of logical space banks, all of which are implemented by a single server process. Each logical bank:

- Allocates and deallocates individual pages and nodes on request.
- Remembers what objects have been allocated from that logical bank, so that they can be bulk-reclaimed when the bank is destroyed.
- Provides locality of allocation so long as this is feasible on the underlying disk, up to the limit of the system-designed extent size.

- Impose optional limits (quotas) on the total number of pages and nodes that can be allocated from that logical bank.
- Provides means to create “child” banks whose storage comes from the parent, creating a hierarchy of storage allocation.

### 5.1 Extent Caching

A naive implementation of the space bank would allocate one object at a time, recording each allocation in some suitable ordered collection. Typically, each dynamically allocated object in the system has associated with it at least one logical bank. For example, most address spaces are implemented as “copy on write” versions of some existing space, and the copied pages (and the nodes that reference them) are allocated from a per-space bank. One impact of this is that space bank invocations are frequent. As a result the single object approach would not provide good locality.

An obvious solution would be to allocate storage to each bank in extents, and allow the bank to suballocate objects from this extent. Unfortunately, this doesn't work well either. If we imagine that each extent contains 64 pages, and that there is some variation in address space sizes, we must conclude that when each address space, process, or other synthesized object has been completely allocated there would remain within its bank a partially allocated extent. In the absence of empirical data, we should expect that this residual extent would on average be half allocated. Unfortunately, there is no simple way to know which banks are done allocating. This means that there would be a very large number of outstanding banks (one per process, one per file, etc.) each of which has committed to it 32 page frames of disk storage that will never be allocated.

The solution to this is *extent caching*. Instead of associating an individual extent with each bank, the space bank maintains a cache (~128 lines) of “active” extents. Each of these extents begins at an OID corresponding to a 64 page boundary on the disk and contains 64 page frames worth of OIDs (some of which may already be allocated). Extent caches are typed: nodes and pages are allocated from distinct extents, which helps to preserve metadata locality. The extent cache design relies on the fact that sequential OIDs correspond with high likelihood to sequential disk locations.

Every bank is associated with a line in the extent cache by a hash on the address of the logical bank data structure. When a bank needs to allocate an object, it first checks availability in its designated cache line. If that extent has no available space, then a “cache miss” occurs, and an

attempt is made to allocate a fresh extent from the underlying disk space. If this proves impossible, as when disk space is near exhaustion, the needed object will be allocated from the first extent in the extent cache that has available space.

The effect of the extent cache is to ensure that banks receive sequential objects in probabilistic fashion up to the limits of the extent size. While it is possible for two banks that are simultaneously active to hash to the same extent, we have not observed it to be a problem. A secondary effect of the extent cache is that the disk page frame allocation map is consulted with reduced frequency. This is desirable because consulting the allocation map involves a linear search through a page and consequently flushes the CPU data cache, which has a significant impact on allocation speed. Our first space bank implementation did this, and we found that the cost of data cache reconstruction after allocation overwhelmed all other costs.

When objects are deallocated, they are restored to the extent cache only if the containing extent is still in the cache. Otherwise, they are returned directly to the free map. Newly freed objects are reused aggressively, because reusing objects that are still in memory eliminates extra disk I/Os that would record their deallocation. Reuse of old objects is deferred. The assumption behind this is that the objects allocated to a given bank share a common temporal extent and will tend to be deallocated as a group. Given this, it is better to wait as long as possible before reusing the available space in an older extent in order to maximize the likelihood that the entire extent has become free.

To support address space metadata locality, the space bank implements a two-level allocation scheme for nodes. The extent cache caches page frames. The space bank sub-allocates nodes sequentially from these frames. Because address spaces are constructed by copy on write methods, and because the copy on write process proceeds top down in the node tree (Figure 2), it is usual for the entire path of nodes from the root to the first page to be allocated from a single page frame on the disk. When the top node is fetched, its containing page frame is cached in the page cache. The effect of this is that the entire sequence of nodes from the address space root node to the referenced page is brought into memory with a single disk I/O.

## 5.2 Record-Keeping Locality

Conceptually, each space bank's record of allocated objects can be kept by any convenient balanced tree structure. The current EROS implementation uses a red-black tree. There are two potential complications that need to be considered in building this tree.

The first is sheer size. Collectively, the number of RB-tree nodes is on the same order as the number of disk objects.

While these structures might fit within the space bank's virtual address space on current machines, they certainly will *not* fit within physical memory in large system configurations. However clever the data structure, care must be taken to ensure that traversals of these structures do not suffer from poor locality due to space bank heap fragmentation. This type of poor locality translates directly into paging. The current space bank implementation does not attempt to manage this issue, which is a potentially serious flaw. A simple solution would be to allocate tree nodes using an extent caching mechanism similar to the one already used for nodes and pages, or a slab-like allocation mechanism [Bon94, BA01].

The second is space overhead. Each OID occupies 64 bits, and it would be disproportionate to spend an additional two or three pointers per object to record allocations. As a result, the current bank tree nodes record extents rather than OIDs, and use a per-extent allocation bitmap to record which objects within an extent have been allocated. Even if two or three banks are simultaneously performing allocations from the same extent cache entry, the net space overhead of this is lower than per-OID recording. The current implementation relies on everything fitting within virtual memory. On the Intel x86 family implementation, this will be adequate until the attached disk space exceeds 2.6 terabytes.

## 6 Two Early Disk-Level Designs

The original EROS system, including its store design, followed the published design of KeyKOS [Har85]. Each disk is divided into *ranges* of sequentially numbered objects. Ranges are partitioned by object type; a given range contains nodes or pages, but not both. Every object's OID consists of a 64 bit "coded disk address" concatenated with a 32 bit "allocation count." The coded disk address describes the location of the object, and the allocation count indicates how many times a particular object has been allocated. In order for a capability to be valid, the allocation count in the capability must match the allocation count recorded on the disk for the corresponding object.

### 6.1 Original Storage Layout

At startup, the kernel probes all disks, identifies the ranges present, and builds an in-memory table with an entry for each range:

(node/page, startOID, endOID, disk, startSec)

It also scans the checkpoint area to rebuild the in-memory directory of object locations.

For nodes, the allocation count is recorded in the node itself. Nodes are numbered sequentially within a range, and are packed in page-sized units called *node pots*; no node on the disk is split across two pages. Once the containing range has been identified, the disk location of the relative disk page frame containing a node can be computed by

$$(OID - startOID) / NodesPerPageFrame$$

For pages, the allocation count cannot be recorded within the page, because all of the available bytes are already in use. Instead, page ranges are further subdivided into subranges. Each subrange begins with an *allocation pot*, which is a page that contains the allocation counts for the following pages in the subrange (Figure 3). The allocation pot also contains a byte containing various flags for the page, including one indicating whether that page is known to hold zeros.

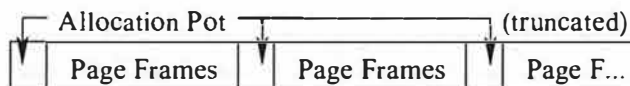


Figure 3: Page range layout.

Each allocation pot can hold information for up to 819 pages, so a page range is organized as a sequence of subranges, each 820 pages long and consisting of an allocation pot followed by its associated pages. Depending on the size of the underlying partition, the final subrange may be truncated. Once the containing range has been identified, the disk location of a relative disk page frame containing a page can be computed by

$$(OID - startOID) + (OID - startOID) / 819 + 1$$

In either case, the relative frame can then be combined with the *startSec* value to yield the starting sector for the I/O.

## 6.2 Unified Object Spaces

The design of Section 6.1 suffers from an irritating flaw: it partitions the disk into typed ranges. There is no easy way to know in advance the correct proportion of nodes to pages, and the design does not provide any simple means to reorganize the disk (there are no forwarding pointers) or to interconvert ranges from one type to another. We found that we were continuously adjusting our directions to the disk formatting program to add or remove objects of some type.

The solution was to adopt the page range layout for all ranges, and use an available bit in the allocation pot to indicate the “type” (node or page) of the corresponding

disk page frame (Figure 4). If the frame type is “page,” the allocation count in the allocation pot is the allocation count of the page, otherwise it is the *max* of the allocation counts of all nodes contained in the frame. The OID encoding was also reorganized, using the least 8 bits as the index of the object within the frame and the upper 56 bits as the “frame OID.” The frame offset computation proceeds as previously described for page frames, with the caveat that the OID value must be shifted before performing the computation.

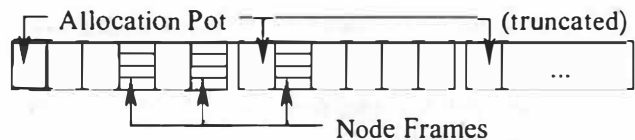


Figure 4: Unified range layout.

In the revised design, the kernel converts a frame from one type to another whenever an object of the “wrong” type is allocated by the space bank. The kernel assumes that the space bank has kept track of available storage, and that it will not unintentionally reallocate storage that is already in use. There is a minor complication, which is that the kernel must ensure that allocation count is never decreased by conversion. This is assured by setting the allocation count to  $\max(node\ alloc\ counts) + 1$  when converting a frame from nodes to pages and setting all node allocation counts to the page allocation count when converting a frame from pages to nodes.

The switch to unified ranges simplifies the kernel object management code, but more importantly it simplifies the use and allocation of disk storage. Disk frames can now be traded back and forth between types as needed. In addition to allowing address space metadata and data to be placed in a localized fashion (thereby facilitating read-ahead across object types), the new design can potentially be extended to perform defragmentation in order to improve extent effectiveness. Individual banks effectively record the relationships between pages, nodes, and their containing objects, and the ability to retype frames supports storage compaction. To perform compaction, two additional bits can be taken from the “flags” field to “lock” an object, copy its content to a new destination frame, use the old frame to record the new location, and then use a second flags bit to mark the object forwarded. Either the space bank or a helper application can now iterate through all nodes, rewriting their capabilities to reflect the new object location.

## 7 Embedded EROS

In early 2000, as part of an exploratory research collaboration with Panasonic, we started to investigate the possibility of an embedded version of EROS for selected real-time applications. As part of this, a decision was made to remove all remaining drivers from the kernel. Together, these decisions introduced three new requirements into the overall system design:

1. In order to support DMA, we needed a way to support pages (but not nodes) whose physical memory address could be known to a driver.
2. The kernel now needed some mechanism for allocation of non-pageable and non-checkpointed objects.
3. Ordinary disk ranges now needed to be served by user-mode drivers.

To address these requirements, the “range” notion was generalized to the notion of *object sources*. An object source implements some sequential range of OIDs. This range may be only partially populated.

By well-known convention, two ranges of OIDs are reserved. One corresponds to physical memory pages, while the other allocates non-pageable objects. EROS already uses main memory as an object cache. The physical page object source will allocate any OID whose range-relative frame index corresponds to a physical memory page frame that is part of the page cache. The effect of allocating a capability for such an OID is to evict the current resident of that page cache entry and relabel the entry as a physical page object. The non-pageable object range is similar, though there is no guarantee that the object will occupy any particular physical address. Both physical memory pages and non-pageable objects are exempted from checkpointing and eviction. When these objects are freed, the corresponding cache locations are returned to the object cache free pool for later reuse.

In the embedded design, the system is partitioned into a non-persistent space that contains drivers and the object store manager, and a persistent space that operates exactly as before. The driver portion of the system is loaded from ROM, and uses an *object source registry* capability to register support for a persistent range if one is to be implemented. The persistent OID range (if present) is “backed” by a user-mode object source driver, and the kernel defines a protocol by which this driver can insert or remove objects whose OIDs fall within the range it controls. When completed, there will also be a protocol by which the persistent source driver indicates how many dirty objects can be permitted for its range at any given time. The implementation of the checkpoint mechanism in this design is relocated to the persistent source driver; the kernel

remains responsible for snapshot and for “writing back” the checkpointed objects to the persistent source driver.

## 8 The Vertical View

As with conventional file systems, the effectiveness of a single-level store design relies on the interaction of temporal, spatial, and referential efficiencies implemented cooperatively by several vertical layers in the system. This section briefly recaps the critical points in a single place so that their combined effect can be more readily seen. Each item is annotated by the section that discusses it.

### *Temporal Efficiency:*

- The kernel ensures that objects that are allocated and deallocated within a single checkpoint interval generate no I/O to home locations provided that capabilities to them are never written to the disk. [4.3]
- The space bank eagerly reuses young, dead objects to reduce unnecessary recording of deallocations, and to aggressively reuse allocation pots that it knows must already be in memory. [5.1]

### *Spatial Efficiency:*

- The space bank allocates objects using bank-wise extents, which helps to preserve disk-level locality. Separate extents are used for pages and nodes. [5.1]
- There is a direct correspondence between OIDs and page frame placement in the store, eliminating the need for directory or indirection blocks in the object store. [6.1,6.2]

### *Referential Efficiency:*

- The address space copy on write implementation combines a dedicated bank (and therefore a dedicated extent) with top-down metadata traversal, ensuring that all “indirect blocks” in a given traversal will tend to be fetched in a single I/O operation. [4.2]
- Both the checkpoint and the migration systems use bulk, sorted I/O, reducing total seek latency in spite of performing a larger number of object writes. [3.3]
- Empty (zero) objects are neither written nor read to the home locations – only their allocation pots are revised. [6.1]
- Both the checkpoint directory [3.2] and the range table [6.1] are kept in memory. No additional disk I/Os are required to determine the location of a target page or node.



The combined effect of this may be illustrated by describing in more detail what happens when an object is to be loaded.

When a page or node is to be fetched, the EROS kernel first consults an in-memory object hash table to determine if the object is already in memory. This includes checking for the containing node pot or allocation pot as appropriate. Next, the checkpoint area directory is consulted to see if the current version of this object is located in the checkpoint area. If the object is not found in the checkpoint area, the range table is consulted and an I/O is initiated for the objects containing page frame and (if needed) its allocation pot. In the typical case, ignoring read-ahead, only one I/O is performed. The effectiveness of the node allocation strategy tends to yield one I/O for every 7 nodes (because 7 nodes fit in a node pot). Similarly, the allocation pot I/O is performed only once for a given 819 frame region in the home locations; the overhead of these I/Os is negligible.

The total effectiveness of the single-level store relies on collaboration between the storage allocator, its clients, the kernel, and the underlying store design. Each of these pieces, taken individually, is relatively straightforward.

Conceptually, this layering is not so different from what happens in a file system. An important difference is that in the file system design this layering is opaque. In EROS, it is straightforward to implement customized memory managers or use multiple space banks for more explicit extent management.

## 9 Related Work

While the idea of a single-level store is widely known among operating system implementors, relatively little has been published about their design. As mentioned in the introduction, the AS/400 is perhaps the best known implementation, but the only widely available reference on this design [So196] provides inadequate details. Even within IBM, information on the AS/400 implementation is closely held.

Both Grasshopper [DdB<sup>+</sup>94] and Mungi [HEV<sup>+</sup>98] use single-level stores. Neither has published details on the storage system itself. Like the AS/400, persistence in the Mungi system is explicitly managed. Grasshopper's is transparent, but its strategy for computing transitive dependencies is both complex and expensive.

Consistent checkpointing has been the subject of several previous papers, most notably work by Elnozahy *et al.* [EJZ92] and Chandy and Lamport [CL85].

KeyKOS, from which the EROS design is derived, uses a single level store and consistent checkpoint mechanism described in [Lan92]. The design of the store itself has

never previously been published.

The L3 system [Lie93] implemented a transparent checkpointing mechanism in its user-level address space manager. Like the KeyKOS and EROS checkpointing designs, this implementation uses asynchronous copy on write for interactive responsiveness. Fluke similarly implemented an experimental system-wide checkpoint mechanism at user level [TLFH96], but this implementation is a "stop and copy" implementation. Disk writes are performed before execution can proceed making the Fluke implementation unsuitable for interactive or real-time applications. Neither the L3 nor the Fluke checkpointers perform any sort of systemwide consistency check prior to writing a checkpoint, introducing the likelihood that system state errors resulting from either imperfect implementation or ambient background radiation will be rendered permanent.

The checkpoint design presented here is similar in many respects to the behavior of log-structured file systems such as LFS [SBMS93, MR92]. As with log-structured file systems, the checkpoint mechanism converts random writes into sequential writes. Unlike the log-structured design, the EROS checkpointing design quickly converts this temporally localized data into physically localized data by migrating it into locations that were allocated based on desired long-term locality. The resulting performance remains faster than conventional file systems, but does not decay as file system utilization increases.

## 10 Future Work

The store designs described in sections 6 and 7 reflect a mature placement strategy that has been tested over a long period of time. While effective, this placement strategy suffers from a significant limitation: it is difficult to administer changes to the underlying disk configuration. While the EROS system implements software duplexing to allow storage to be rearranged, the rearrangement process is neither efficient nor "hands free." It would be better to have an automated means to take advantage of new and larger stores.

The current EROS space bank implementation can theoretically handle stores slightly larger than  $2^{29}$  pages (2.6 terabytes). This corresponds approximately to one fully-populated RAID subsystem containing 10 drives, each providing 60 gigabytes of storage. While not common on the desktop, these configurations appear more and more frequently on servers. The paging behavior of the current space bank implementation would be quite bad for this size store. While the space bank could be reimplemented to eliminate thrashing during allocation, a better approach overall would be to loosen the association between OIDs and disk page frames. Extent-level locality



is essential, but a sparsely allocatable OID space would eliminate the need for the red-black trees that are currently used to record object allocations.

From an addressing standpoint, larger stores do not present an immediate problem for EROS. The underlying OID space can handle stores of up to  $2^{56}$  pages.<sup>4</sup> Given that there is no basic addressability problem, the key question is “how will we manage the growth?” The sequential allocation strategy used by the space bank does not do an effective job of balancing load over disk arms in the absence of a RAID controller. Further, the current mapping strategy from OIDs to disk page frames does not lend itself to physical rearrangement of objects as the available storage grows. All of these issues point to a need for a logical volume mechanism.

The next and (we hope) final version of the EROS single level store will likely be based on randomization-based extent placement. This is a nearly complete departure from the designs described here:

- Node and page OID spaces are once again partitioned.
- Allocation counts are abandoned. OIDs can be re-allocated only if the space bank knows that no capability using the OID exists on the disk.
- The direct map from OIDs to disk ranges has been abandoned entirely. Objects are now placed using a randomization-based strategy.
- While extent-based object placement continues to be honored on a “best effort” basis using low-order bits of the OID, there is no longer any direct association between an OID and the location of an object on disk.
- Ranges can be dynamically grown or shrunk as disks are added and removed.
- Where the previous operations on ranges were “allocate” and “deallocate,” the new design separates object allocation into two parts: storage reservation and object name binding.

The inspiration for this departure is a new, randomization-based disk placement strategy being explored within the Systems Research Laboratory based on prior work by Brinkmann and Scheideler [BSS00]. We plan to adopt a single-disk variant of this strategy in the EROS single-level store.

The key motivation for this change is the ability to divorce OIDs from physical placement without losing the ability to directly compute object addresses. Under the

<sup>4</sup> The disk drive industry has yet to produce  $2^{56}$  pages of total disk storage over the lifespan of the industry, but will soon cross this mark.

new strategy, disks or partitions can be added or removed from the system at will without needing to garbage collect or renumber OIDs, and data can be transparently shifted to balance load across available disk arms. This renders the system more easily scalable, and provides the type of load balancing and latency properties needed for multimedia applications.

## 11 Acknowledgements

The KeyKOS single level store was designed by Norm Hardy and Charles Landau while at Tymshare, Inc. Both have been helpful and patient in describing the workings of KeyKOS and encouraging the development and evolution of the EROS system. Bryan Ford was kind enough to explain the Fluke checkpointing implementations in some detail.

Jochen Liedtke similarly took time to explain the L3 checkpointing implementations. Jochen’s continuous advances in the performance and design of microkernel operating systems led to improvements in the EROS implementation and drove us to a deeper and more careful understanding of operating system design.

## 12 Conclusions

This paper describes three working implementations of a single-level store. To our knowledge, this is the first time that any single-level store design has been comprehensively described in the public literature. The code is available online, and can be found at the EROS web site [Sha]. In describing our design, we have attempted to identify both the critical performance issues that arise in single level store designs and the solutions we have found to those issues.

One key to an effective single-level store is the interaction between temporal, spatial, and referential efficiency. This is made possible in EROS by the fact that disk-level locality information is rendered directly available for application use. Where file systems make locality decisions at the time the file is closed and the file cache is *flushed*, the EROS space bank makes these decisions when the corresponding storage is *allocated*, which is the point where maximal semantic knowledge of intended usage is at hand. An interesting challenge in the randomization-based design is to preserve an effective balance between spatial locality and adaptive scalability.

A surprising attribute of the EROS single level store is that in spite of its vertical integration it has undergone several major changes with minimal application-level impact. We have changed the capability size, the OID encoding, the checkpoint design, and removed the object driver from

the kernel. The only application code that changed was the space bank. Within the kernel itself, even the cache management code has gone largely unchanged as these modifications occurred.

From a design perspective, this paper has illustrated that single level stores simplify operating system design by removing an unnecessary layer of abstraction from the system. Instead of implementing a new and different semantics at the file system layer, a single level store extends the memory mapping model downwards to include the disk, allowing applications to control placement directly. In EROS these placement controls are generally provided by standard fault handling programs; most applications simply use these handlers, and require no code at all for storage management – separation of concerns is effectively maintained. On the other hand, applications with unusual requirements can replace these fault handlers if needed. The total EROS system size is roughly 25% that of Linux.

Microbenchmarks [SSF99] show that performance-critical object allocations in EROS are fast. Hand examination shows that the mechanisms described here are actually generating good disk-level locality. EROS-specific benchmarks show that EROS makes effective use of the available sustained disk bandwidth. In practice the main problem with checkpointing seems to finding a heuristic that does the associated I/Os *slowly* enough to avoid interfering with interactive processing. We also know that the KeyKOS database system, whose disk performance is critical, delivered exceptionally strong performance. All this being said, a key missing piece in this paper is application-level benchmarks. We are in the process of porting several server and client applications to EROS, and plan to measure application-level performance when this has been done.

*This paper is dedicated to the memory of Prof. Dr. Jochen Liedtke (1953–2001).*

### 13 About the Authors

Jonathan S. Shapiro is an Assistant Professor in the Computer Science department of Johns Hopkins University. His current research interests include secure operating systems, high-assurance software development, and adaptive storage management. He is also a key participant in the Hopkins Information Security Institute.

Jonathan Adams is a recent graduate of the California Institute of Technology, and is now a Member of Technical Staff in the Solaris Kernel Development group at Sun Microsystems.

### References

- [BA01] Jeff Bonwick and Jonathan Adams. Magazines and vmem: Extending the slab allocator to many cpu's and arbitrary resources. In *Proc. 2001 USENIX Technical Conference*. USENIX Association, 2001.
- [BC00] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Press, October 2000.
- [BHK<sup>+</sup>91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery SIGOPS, 1991.
- [Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.
- [BSS00] A. Brinkmann, K. Salzwedel, and C. Scheider. Efficient, distributed data placement strategies for storage area networks. In *Proc. of the 12th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 119–128, 2000.
- [CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.
- [DdB<sup>+</sup>94] Alan Dearl, Red di Bona, James Farrow, Frans Henskens, Anders Lindstrom, John Rosenberg, and Francis Vaughn. Grasshopper: An orthogonally persistent operating system. *Computer Systems*, 7(3):289–312, 1994.
- [Dv66] J. B. Dennis and E. C. van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–154, March 1966.
- [EJZ92] Elmootazbellah Nabil Elnozahy, David B. Johnson, and Willy Zwaenpoel. The Performance of Consistent Checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, Houston, Texas, 1992.
- [Har85] Norman Hardy. The KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, October 1985.

- [HEV<sup>+</sup>98] Gernot Heiser, Kevin Elphinstone, Jerry Vochtelloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software – Practice and Experience*, 28(9):901–928, 1998.
- [IBM98] *AS/400 Machine Internal Functional Reference*. Number SC41-5810-01. IBM Corporation, 1998.
- [Lan92] Charles R. Landau. The checkpoint mechanism in KeyKOS. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, pages 86–91. IEEE, September 1992.
- [Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 175–188. ACM, 1993.
- [MJLF84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.
- [MR92] J. Ousterhout M. Rosenblum. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, (1):26–52, February 1992.
- [RW93] Chris Ruemmler and John Wilkes. Unix disk access patterns. In *Proc. USENIX Winter 1993 Technical Conference*, pages 405–420, San Diego, California, January 1993.
- [SBMS93] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 307–326, San Diego, CA, USA, 25–29 1993.
- [SFS96] Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. State caching in the EROS kernel – implementing efficient orthogonal persistence in a pure capability system. In *Proc. 7th International Workshop on Persistent Object Systems*, pages 88–100, Cape May, NJ, USA, 1996.
- [SG99] Jiri Schindler and Gregory R. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, November 1999.
- [Sha] Jonathan S. Shapiro. *The EROS Web Site*. <http://www.eros-os.org>.
- [Sol96] Frank G. Soltis. *Inside the AS/400*. Duke Press, Loveland, Colorado, 1996.
- [SS95] Keith A. Smith and Margo I. Seltzer. File system aging – increasing the relevance of file system benchmarks. In *Proc. 1995 USENIX Technical Conference*, pages 249–264, New Orleans, LA, USA, January 1995.
- [SSB<sup>+</sup>95] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McManis, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proc. 1995 USENIX Technical Conference*, pages 249–264, New Orleans, LA, USA, January 1995.
- [SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.
- [TLFH96] Patrick Tullmann, Jay Lepreau, Bryan Ford, and Mike Hibler. User-level checkpointing through exportable kernel state. In *Proc. 5th IEEE International Workshop on Object-Oriented in Operating Systems*, pages 85–88, October 1996.

# THINK: A Software Framework for Component-based Operating System Kernels

Jean-Philippe Fassino  
*France Télécom R&D*

jeanphilippe.fassino@francetelecom.com

Jean-Bernard Stefani  
*INRIA*

jean-bernard.stefani@inria.fr

Julia Lawall  
*DIKU*

julia@diku.dk

Gilles Muller  
*INRIA\**

gilles.muller@emn.fr

## Abstract

Building a flexible kernel from components is a promising solution for supporting various embedded systems. The use of components encourages code re-use and reduces development time. Flexibility permits the system to be configured at various stages of the design, up to run time. In this paper, we propose a software framework, called THINK, for implementing operating system kernels from components of arbitrary sizes. A unique feature of THINK is that it provides a uniform and highly flexible *binding model* to help OS architects assemble operating system components in varied ways. An OS architect can build an OS kernel from components using THINK without being forced into a predefined kernel design (e.g. exo-kernel, micro-kernel or classical OS kernel). To evaluate the THINK framework, we have implemented KORTEx, a library of commonly used kernel components. We have used KORTEx to implement several kernels, including an L4-like micro-kernel, and kernels for an active network router, for the Kaffe Java virtual machine, and for a Doom game. Performance measurements show no degradation due to componentization and the systematic use of the binding framework, and that application-specific kernels can achieve speed-ups over standard general-purpose operating systems such as Linux.

## 1 Introduction

Embedded systems, such as low-end appliances and network routers, represent a rapidly growing domain of systems. This domain exhibits specific characteristics that impact OS design. First, such systems run one or only a small family of applications with specific needs. Second, for economic reasons, memory and CPU resources

are scarce. Third, new hardware and software appear at a rapid rate to satisfy emerging needs. Finally, systems have to be flexible to support unanticipated needs such as monitoring and tuning.

Implementing an operating system kernel for such systems raises several constraints. Development time should be as short as possible; this encourages systematic *code re-use* and implementation of the kernel by assembling existing components. Kernel size should be minimal; only services and concepts required by the applications for which the kernel is targeted should be embedded within the kernel. Efficiency should be targeted; no specific hardware feature or low-level kernel functionality should be masked to the application. Finally, to provide flexibility, it must be possible to instantiate a kernel configuration at boot time and to dynamically download a new component into the kernel. To support these features, it should be possible to resolve the bindings between components at run time.

Building flexible systems from components has been an active area of operating system research. Previous work includes micro-kernel architectures [2, 14, 25], where each component corresponds to a domain boundary (i.e. server), extensible systems such as SPIN [3] that support dynamic loading of components written in a type-safe language, and more recently the OSKit [7] or eCos [5] which allow the re-use of existing system components. One problem with the existing component-based approaches lies in the predefined and fixed ways components can interact and be bound together. While it is possible to create specific components to implement a particular form of binding between components, there are no supporting framework or tools to assist this development, and little help to understand how these different forms of binding can be used and combined in a consistent manner.

By *binding* we mean the end result of the general process of establishing communication or interaction channels between two or more objects. Bindings may take sev-

\*Now at Ecole des Mines de Nantes

eral forms and range from simple pointers or in-address-space references, to complex distributed channels established between remote objects involving multiple layers of communication protocols, marshalling and unmarshalling, caching and consistency management, etc. As argued in [27], the range and forms of bindings are so varied that it is unlikely that a single generic binding process or binding type can be devised. This in turn calls for framework and tool support to help system developers design and implement specific forms of binding.

The attention to flexible binding is not new. Several works, e.g. [4, 10, 27], have proposed flexible binding models for distributed middleware platforms. The Nemesis operating system [13] introduces bindings for the handling of multimedia communication paths. The path abstraction in the Scout operating system [19] can be understood as an in-system binding abstraction. And channels in the NodeOS operating system interface for active routers [22] correspond to low-level packet-communication-oriented bindings. None of these works, however, has considered the use of a general model of component binding as a framework for building different operating system kernels.

## This paper

This paper presents the THINK<sup>1</sup> framework for building operating system kernels from components of arbitrary sizes. Each entity in a THINK kernel is a component. Components can be bound together in different ways, including remotely, through the use of bindings. Bindings are themselves assemblies of components that implement communication paths between one or more components. This structuring extends to the interaction with the hardware, which is encapsulated in Hardware Abstraction Layer (HAL) components.

The contributions of this paper are as follows. We propose a software architecture that enables operating system kernels to be assembled, at boot-time or at run-time, from a library of kernel components of arbitrary size. The distinguishing feature of the framework is its flexible binding model that allows components to be bound and assembled in different and non-predefined ways.

We have designed and implemented KORTEx, a library of kernel components that offers several models of threads, memory management and process management services. KORTEx implements different forms of bindings, including basic forms such as system calls

(syscalls), up-calls, signals, IPC and RPC calls. We have used KORTEx to implement L4-like kernel services. Our benchmarks show excellent performance for low-level system services, confirming that applying our component model and our binding model does not result in degraded performance compared to non component-based kernels.

We have used KORTEx to implement operating system kernels for an active network router, the Kaffe Java virtual machine, and a Doom game. We have evaluated the performance of these kernels on a Macintosh/PowerPC machine. Our benchmarks show that our kernels are at least as efficient than the implementations of these applications on standard monolithic kernels. Additionally, our kernels achieve small foot-prints. Finally, although anecdotal, our experience in using the THINK framework and the KORTEx library suggests interesting benefits in reducing the implementation time of an operating system kernel.

The rest of the paper is structured as follows. Section 2 discusses related work on component-based kernels and OSes. Section 3 details the THINK software framework, its basic concepts, and its implementation. Section 4 describes the KORTEx library of THINK components. Section 5 presents several kernels that we have assembled to support specific applications and their evaluation. Section 6 assesses our results and concludes with future work.

## 2 Related Work

There have been several works in the past decade on flexible, extensible, and/or component-based operating system kernels. Most of these systems, however, be they research prototypes such as Choices and  $\mu$ -Choices [31], SPIN [3], Aegis/Xok [6], VINO [26], Pebble [8], Nemesis [13], 2K [12], or commercial systems such as QNX [23], VxWorks [34], or eCos [5], still define a particular, fixed set of core functions on which all of the extensions or components rely, and which implies in general a particular design for the associated family of kernels (e.g. with a fixed task or thread model, address space model, interrupt handling model, or communication model). QNX and VxWorks provide optional modules that can be statically or dynamically linked to the operating system, but these modules rely on a basic kernel and are not designed according to a component-based approach. eCos supports the static configuration of components and packages of components into embed-

<sup>1</sup> THINK stands for Think Is Not a Kernel

ded operating systems but relies on a predefined basic kernel and does not provide dynamic reconfiguration capabilities.

In contrast, the THINK framework does not impose a particular kernel design on the OS architect, who is free to choose e.g. between an exo-kernel, a micro-kernel or a classical kernel design, a single or multiple address space design. In this respect, the THINK approach is similar to that of OSKit [7], which provides a collection of (relatively coarse-grained, COM-like) components implementing typical OS functionalities. The OSKit components have been used to implement several highly specialized OSes, such as implementations of the programming languages SML and Java at the hardware level [15]. OSKit components can be statically configured using the Knit tool [24]. The Knit compiler modifies the source code to replace calls across component boundaries by direct calls, thus enabling standard compiler optimizations. Unlike THINK however, OSKit does not provide a framework for binding components. As a result, much of the common structures which are provided by the THINK framework have to be hand-coded in an ad-hoc fashion, hampering composition and reuse. Besides, we have found in practice that OSKit components are much too coarse-grained for building small-footprint, specific kernels that impose no particular task, scheduling or memory management model on applications. Other differences between THINK and OSKit include:

- **Component model:** THINK has adopted a component model inspired by the standardized Open Distributed Processing Reference Model (ODP) [1], whereas OSKit has adopted Microsoft COM component model. While the two component models yield similar run-time structures, and impose as few constraints on component implementations, we believe that the THINK model, as described in section 3.1 below, provides more flexibility in dealing with heterogeneous environments.
- **Legacy code:** OSKit provides several libraries that encapsulate legacy code (e.g. from FreeBSD, Linux, and Mach) and has devoted more attention to issues surrounding the encapsulation of legacy code. In contrast, most components in the KORTEx library are native components, with the exception of device drivers. However, techniques similar to those used in OSKit (e.g. emulation of legacy environments in glue code) could be easily leveraged to incorporate in KORTEx coarse-grained legacy components.
- **Specialized frameworks:** in contrast to OSKit,

the KORTEx library provides additional software frameworks to help structure kernel functionality, namely a resource management framework and a communication framework. The resource management framework is original, whereas the communication framework is inspired by the *x*-kernel [11].

Other operating system-level component-based frameworks include Click [18], Ensemble [15] and Scout [19]. These frameworks, however, are more specialized than THINK or OSKit: Click targets the construction of modular routers, Ensemble and Scout target the construction of communication protocol stacks.

We thus believe that THINK is unique in its introduction and systematic application of a flexible binding model for the design and implementation of component-based operating system kernels. The THINK component and binding models have been inspired by various works on distributed middleware, including the standardized ODP Reference Model [1], ANSA [10], and Jonathan [4]. In contrast to the latter works, THINK exploits flexible binding to build operating system kernels rather than user-level middleware libraries.

### 3 THINK Software Framework

The THINK software framework is built around a small set of concepts, that are systematically applied to build a system. These concepts are: *components*, *interfaces*, *bindings*, *names* and *domains*.

A THINK system, i.e. a system built using the THINK software framework, is composed of a set of domains. *Domains* correspond to resource, protection and isolation boundaries. An operating system kernel executing in privileged processor mode and a set of user processes executing in unprivileged processor mode are examples of domains. A domain comprises a set of components. Components interact through bindings that connect their interfaces. Domains and bindings can themselves be reified as components, and can be built by composing lower-level components. The syscall bindings and remote communication bindings described in section 4 are examples of composite bindings, i.e. bindings composed of lower-level components. Bindings can cross domain boundaries and bind together interfaces that reside in different domains. In particular, components that constitute a composite binding may belong to different domains. For example, the aforementioned syscall and remote communication bindings cross domain boundaries.



### 3.1 Core software framework

The concepts of *component* and *interface* in the THINK framework are close to the concepts of object and interface in ODP. A component is a run-time structure that encapsulates data and behavior. An interface is a named interaction point of a component, that can be of a server kind (i.e. operations can be invoked on it) or of a client kind (i.e. operations can be invoked from it). A component can have multiple interfaces. A component interacts with its environment, i.e. other components, only through its interfaces. All interfaces in THINK are strongly typed. In the current implementation of the THINK framework, interface types are defined using the Java language (see section 3.2). Assumptions about the interface type system are minimum: an interface type documents the signatures of a finite set of operations, each operation signature containing an operation name, a set of arguments, a set of associated results (including possible exceptions); the set of interface types forms a lattice, ordered by a subtype relation, allowing multiple inheritance between interface types. The strong typing of interfaces provides a first level of safety in the assembly of component configurations: a binding can only be created between components if their interfaces are type compatible (i.e. are subtypes of one another).

An interface in the THINK framework is designated by a *name*. Names are context-dependent, i.e. they are relative to a given *naming context*. A naming context encompasses a set of created names, a naming convention and a name allocation policy. Naming contexts can be organized in naming graphs. Nodes in a naming graph are naming contexts or other components. An edge in a naming graph is directed and links a naming context to a component interface (which can be another naming context). An edge in a naming graph is labelled by a name: the name, in the naming context that is the edge source, of the component interface that is the edge sink. Given a naming graph, a naming context and a component interface, the name of the component interface in the given naming context can be understood as a path in the naming graph leading from the naming context to the component interface. Naming graphs can have an arbitrary forms and need not be organized as trees, allowing new contexts to be added to a naming graph dynamically, and different naming conventions to coexist (a crucial requirement when dealing with highly heterogeneous environments as may be the case with mobile devices).

Interaction between components is only possible once a *binding* has been established between some of their interfaces. A binding is a communication channel be-

tween two or more components. This notion covers both language-level bindings (e.g. associations between language symbols and memory addresses) as well as distributed system bindings (e.g. RPC or transactional bindings between clients and possibly replicated servers). In the THINK framework, bindings are created by special factory components called *binding factories*. A binding typically embodies communication resources and implements a particular communication semantics. Since several binding factories may coexist in a given THINK system, it is possible to interact with a component according to various communication semantics (e.g. local or remote; standard point-to-point at-most once operation invocation; component invocation with monitoring, with access control, with caching; event casting à la SPIN; etc). Importantly, bindings can be created either implicitly, e.g. as in standard distributed object systems such as Java RMI and CORBA where the establishment of a binding is hidden from the component using that binding, or explicitly, i.e. by invocation of a binding factory. Explicit bindings are required for certain classes of applications such as multimedia or real-time applications, that impose explicit, application-dependent quality of service constraints on bindings. Creating a binding explicitly results in the creation of a binding component, i.e. a component that reifies a binding. A binding component can in turn be monitored and controlled by other components.

```
interface Top { }
interface Name {
    NamingContext getNC();
    String toByte();
}
interface NamingContext {
    Name export(Top itf, char[] hint);
    Name byteToName(String name);
}
interface BindingFactory {
    Top bind(Name name, char[] hint);
}
```

Figure 1: Framework for interfaces, names and bindings

These concepts of naming and binding are manifested in the THINK software framework by the set of Java interface declarations shown in Figure 1. The type *Top* corresponds to the greatest element of the type lattice, i.e. all interface types are a subtype of *Top* (all interface types in THINK “extend” *Top*). The type *Name* is the supertype of all names in THINK. The operation *getNC* yields the naming context to which the name belongs (i.e. the naming context in which the name has been created through the *export* operation). The operation *toByte* yields a simple serialized form of the

name instance.

The type `NamingContext` is the supertype of all naming contexts in THINK. The operation `export` creates a new name, which is associated to the interface passed as a parameter (the hint parameter can be used to pass additional information, such as type or binding data, required to create a valid name). As a side-effect, this operation may cause the creation of (part of) a binding with the newly named interface (e.g. creating a server socket in a standard distributed client-server setting). The operation `byteToName` returns a name, upon receipt of a serialized form for that name. This operation is guaranteed to work only with serialized forms of names previously exported from the same naming context. The type `NamingContext` sets minimal requirements for a naming context in the framework. More specific forms of naming contexts can be introduced if necessary as subtypes of `NamingContext` (e.g. adding a `resolve` operation to traverse a naming graph).

The type `BindingFactory` is the supertype of all binding factories in THINK. The operation `bind` creates a binding with the interface referenced by the name passed as a parameter (the hint parameter can be used to pass additional information required to establish the binding, e.g. type or quality of service information). Actual binding factories can typically add more specialized `bind` operations, e.g. adding parameters to characterize the quality of service required from the binding or returning specific interfaces for controlling the newly constructed binding.

### 3.2 Implementing the THINK framework

In our current prototype, THINK components are written in C for efficiency reasons. An interface is represented by an interface descriptor structure, whose actual size and content are unknown to the client, and which contains a pointer to the code implementing the interface operations, as shown, in figure 2. This layout is similar to a C++ virtual function table.

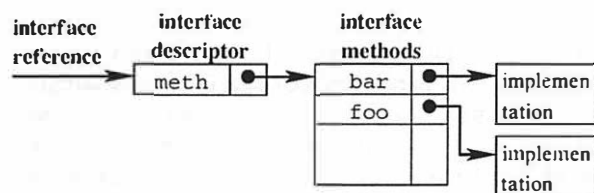


Figure 2: Run-time interface representation

The exact location of private component data is the responsibility of the component developer. Depending on the nature of the target component, the implementation supports several optimizations of the structure of the interface representation. These optimizations help reduce, for instance, memory and allocation costs when handling interface descriptors. They are depicted in figure 3. If the component is a singleton, i.e. there is no other component in the given domain implementing the same interface, then the interface descriptor and the component private data can be statically allocated by the compiler. If the component is not a singleton but has only one interface, then the private data of the component can be allocated directly with the interface descriptor. Finally, in the general case, the interface descriptor is a dynamic structure containing a pointer to the interface operations and an additional pointer to the component's private data. In the component library described in section 4, most components are either singletons or have a single interface, and are implemented accordingly.

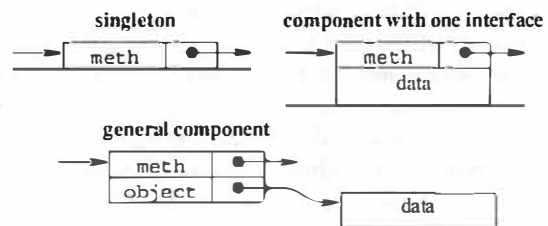


Figure 3: Optimization on interface representation

This implementation abides by C compiler ABI calling conventions [35]. Thus, arguments in a PowerPC implementation are passed on the stack and in registers to improve performance. It is important to notice that all calls to a component interface are expressed in the same way, regardless of the underlying binding type and the location of the component. For example, a server component in the local kernel domain is called in the same way as a server component in a remote host; only the binding changes<sup>2</sup>.

### 3.3 Code generation and tools

Building a particular kernel or an application using the THINK framework is aided by two main off-line tools.

- An open interface compiler, that can be specialized

<sup>2</sup>This does not mean that the client code need not be prepared to handle the particular semantics associated with a binding, e.g. handling exceptions thrown by a remote binding component in case of communication failures.

to generate code from interface descriptions written in Java. For instance, it is used to generate C declarations and code that describe and produce interface descriptors, and to generate components (e.g. stub components) used by binding factories to create new bindings. This generated code can contain assembly code and exploit the specific features of the supporting hardware.

- An off-line configurator, that creates kernel images by assembling various component and binding libraries. This tool implicitly calls a linker (such as `ld`) and operates on a component graph specification, written in UML by kernel developers, which documents dependencies between components and component libraries. Dependencies handled by the configurator correspond to classical functional component dependencies resulting from *provides* and *requires* declarations (*provides* means that a component supports an interface of the given type, *requires* mean that a component requires an interface of the given type to be present in its environment in order to correctly operate). An initialization scheduler, analogous to the OSKit's Knit tool [24], can be used to statically schedule component initialization (through calls to component constructors) at boot-time. The configurator also includes a visual tool to browse composition graphs.

Using the open interface compiler, interface descriptions written in Java are mapped onto C declarations, where Java types are mapped on C types. The set of C types which are the target of this mapping constitutes a subset of possible C signatures. However, we have not found this restriction to be an impediment<sup>3</sup> for developing the KORTEx library.

Code generation takes place in two steps. The first step compiles interface descriptions written in Java into C declarations and code for interface descriptors. These are then linked with component implementation code. Binding components are also generated from interface descriptions but use a specific interface compiler (typically, one per binding type), built using our open interface compiler. The second step assembles a kernel image in ELF binary format from the specification of a component graph.

During execution, a kernel can load a new component, using the KORTEx dynamic linker/loader, or start a new application, by using the KORTEx application loader.

<sup>3</sup>Note that, if necessary, it is always possible to specialize the open interface compiler to map designated Java interface types onto the required C types.

## 4 KORTEx, a component library

To simplify the development of operating system kernels and applications using THINK, we have designed a library of services that are commonly used in operating system construction. This library, called KORTEx, is currently targeted for Apple Power Macintoshes<sup>4</sup>. KORTEx currently comprises the following major components:

- HAL components for the PowerPC that reify exceptions and the memory management unit.
- HAL components that encapsulate the Power Macintosh hardware devices and their drivers, including the PCI bus, the programmable interrupt controller, the IDE disk controller, the Ethernet network card (mace, bmac, gmac and Tulip), and the graphic card (frame-buffer).
- Memory components implementing various memory models, such as paged and flat memory.
- Thread and scheduler components implementing various scheduler policies, such as cooperative, round-robin and priority-based.
- Network components, architected according to the *x*-kernel communication framework, including Ethernet, ARP, IP, UDP, TCP and SunRPC protocols.
- File system components implementing the VFS API, including ext2FS and NFS.
- Service components that implement a dynamic linker/loader, an application loader and a small trader.
- Interaction components that provide different types of bindings.
- Components implementing part of the Posix standard.

While many of these components are standard (for instance, the thread and memory components have been directly inspired by the L4 kernel [9]), several points about KORTEx are worth noting. First, KORTEx systematically exploits the core THINK framework presented above. In particular, KORTEx interaction components presented in section 4.5 all conform to the THINK binding model. The diversity of interaction semantics already available is a testimony to the versatility of this

<sup>4</sup>The choice of PowerPC-based machines may seem anecdotal, but a RISC machine does offer a more uniform environment for operating system design.

model. Second, KORTEx remains faithful to the overall THINK philosophy which is to not impose specific design choices to the OS architect. This is reflected in the fact that most KORTEx components are very fine-grained, including interaction components. For instance, syscall bindings (whose structure and semantics are typically completely fixed in other approaches) are built as binding components in KORTEx. Another example can be found with the HAL components in KORTEx, which strictly reflect the capabilities of the supporting hardware. Third, KORTEx provides additional optional frameworks to help OS architects assemble specific subsystems. KORTEx currently provides a resource management framework and a communication framework. The former is applied e.g. to implement the thread and scheduling components, while the latter is applied to implement remote bindings. Finally, we have strived in implementing KORTEx to minimize dependencies between components. While this is more a practical than a design issue, we have found in our experiments that fine-grained, highly independent components facilitate comprehension and reuse, while obviously yielding more efficient kernels, with smaller footprints. This is an advantage compared to the current OSKit library, for instance.

#### 4.1 HAL components for the PowerPC

KORTEx provides HAL components for the PowerPC, including a HAL component for PowerPC exceptions and a HAL component for the PowerPC Memory Management Unit (MMU). The operations supported by these components are purely functional and do not modify the state of the processor, except on explicit demand.

The KORTEx HAL components manifest strictly the capabilities of the supporting hardware, and do not try to provide a first layer of portability as is the case, e.g. with  $\mu$ Choices' nano-kernel interface [31].

##### Exceptions

The PowerPC exceptions HAL component supports a single interface, which is shown in Table 4. The goal of this interface is to reify exceptions efficiently, without modifying their semantics. In particular, note that, on the PowerPC, processing of exceptions begins in supervisor mode with interrupts disabled, thus preventing recursive exceptions.

When an exception *id* occurs, the processor invokes

```
interface Trap {
    void TrapRegister(int id, Handler handler);
    void TrapUnregister(int id);
    void TrapSetContext(int phyctx, Context virtctx);
    Context TrapGetContext();
    void TrapReturn();
}
```

Figure 4: Interface for PowerPC exception

one of the internal component methods `TrapEnterid`. There is an instance of this method in each exception vector table entry. These methods first save the general registers, which form the *minimal execution context* of the processor, at a location previously specified by the system using the method `TrapSetContext`. This location is specified by both its virtual and physical addresses, because the Power PC exceptions HAL component is not aware of the memory model used by the system. `TrapEnterid` also installs a stack for use during the handling of the exception. A single stack is sufficient for exception handling because the processor disables interrupts during the handling of an exception. Next, `TrapEnterid` invokes the handler previously registered by the system using the method `TrapRegister`. When this handler finishes, the handler calls `TrapReturn` to restore the saved execution context. The cost for entering and returning from an exception on a PowerPC G4 running at 500 Mhz is shown on table 1.

Operation	instructions	time ( $\mu$ s)	cycles
<code>TrapEnter<sub>id</sub></code>	57	0.160	80
<code>TrapReturn</code>	48	0.110	55
total	105	0.270	135

Table 1: Cost for handling a exception

Although minimal, this interface provides enough functionality, e.g. to directly build a scheduler, as shown in section 4.4. The exceptions HAL component is completely independent of the thread model implemented by the system that uses its service.

##### Memory Management Unit

The PowerPC Memory Management Unit (MMU) HAL component implements the software part of the PowerPC MMU algorithm. This component can be omitted in appliances that need only flat memory. Table 5 shows the interface exported by this component.

```

interface MMU {
    void MMUsetpagetable(int virt,
                        int phys, int sz);
    void MMUaddmapping(int vsid, int virt,
                    int phys, int wimg, int pp);
    void MMUremovemapping(int vsid, int virt);
    PTE MMUgetmapping(int vsid, int virt);
    void MMUsetsegment(int vsid, int vbase);
    void MMUsetbat(int no, int virt, int phys,
                int size, int wimg, int pp);
    void MMUremovebat(int no);
}

```

Figure 5: Interface for PowerPC MMU

The `MMUsetpagetable` method is used to specify the location of the page table in memory. Since the PowerPC is a segmented machine, the `MMUsetsegment` method is used to set the sixteen 256 MB segments, thus providing a 4 GB virtual address space. The `MMUaddmapping`, `MMUremovemapping` and `MMUgetmapping` methods add, remove and obtain information about page translation.

The methods `MMUsetbat` and `MMUremovebat` reify the PowerPC Block Address Translation (BAT) registers. These registers provide a convenient way to build a single flat address space, such as can be used in low-end appliances. The two main benefits are speed of address translation and economy of page table memory use.

## 4.2 Resource management framework

KORTEX provides a resource management framework which can be applied to all resources in the system at various levels of abstraction. The framework comprises the resource and manager concepts as given in Figure 6. A resource manager controls lower-level resources and uses them to construct higher-level ones. New resources (e.g. threads) can be created through operation `create`, whereas resource allocation is effected through the `bind` operation which creates a binding to a given resource. In other words, a resource is allocated to a component when a binding has been created by the resource manager between the component and the resource. In this case, the hint parameter of the `bind` operation can contain managing information associated with the resource (e.g. scheduling parameters to be associated with a thread).

Several KORTEX components are architected according to the resource framework: threads and schedulers,

```

interface AbstractResource {
    void release();
}
interface ResourceManager extends BindingFactory {
    AbstractResource create(...);
}

```

Figure 6: Resource management framework

memory and memory managers, network sessions (resources) and protocols (resource managers).

## 4.3 Memory management components

KORTEX provides memory management components that implement various memory models, such as paged memory and flat memory. A paged memory model can be used by systems that need multiple address spaces, for example to provide a process abstraction. The flat memory component can be used by systems that need only a kernel address space, as can be the case e.g. in low-end appliances. KORTEX also provides a component that implements the standard C allocator. Components implementing the two memory models and the allocator are described below.

The flat memory components implement a single kernel address space component that includes all of physical memory. This address space is provided by using `MMUsetbat` exported by MMU HAL (see Section 4.1). This component supports an address space interface providing methods to map and unmap memory in this address space. The implementation of this component is essentially void but the address space interface it supports is useful to provide a transparent access to memory for components, such as drivers, that need to map memory and that can operate similarly with either flat memory or paged memory.

Components providing the paged memory create, during initialization, a page table in memory and an address space for the kernel. An address space manager component provides an interface for creating new address space components. Address space components support interfaces of the same type as that of the flat memory address space component. Physical memory page allocation is provided by a standard buddy system component.

Finally, a dynamic memory allocator component provides the implementation of the standard GNU memory allocator.

## 4.4 Thread and scheduler components

KORTEX provides three preemptive schedulers that provide an same interface of the same type: a cooperative scheduler, a simple round-robin scheduler and a priority-based scheduler. They allow the usual operations on threads: creating and destroying threads, as well as allowing a thread to wait on a condition and to be notified. If threads are not running in the same address space, then the scheduler performs the necessary address space switch in addition to the thread context switch.

These schedulers are implemented using the PowerPC exceptions HAL component described in section 4.1. They can be implemented by simply installing a timer interrupt handler, in fact the PowerPC decremter. On a decremter exception, the handler uses `TrapSetContext` to replace the pointer to the execution context of the current thread with a pointer to the execution context of the newly scheduled thread. Due to the simplicity of the HAL, these schedulers can be very efficient. Table 2 presents context switching costs on a PowerPC G4 at 500 Mhz. For example, a context switch between two threads in the same address space costs 0.284  $\mu$ s, and between two threads in different address spaces costs 0.394  $\mu$ s. This permits the use of extremely small time slices, which can be useful e.g. for a real-time kernel.

	instructions	time ( $\mu$ s)	cycles
thread switch	111	0.284	142
process switch	147	0.394	197

Table 2: Context switching costs

## 4.5 Interaction components

KORTEX provides many different types of bindings between components, which may be localized in different domains (e.g. the kernel, an application, or a remote host).

### Local binding

This binding type is the simplest form of binding and is used for interactions between components in the same domain. It is implemented by a simple pointer to an interface descriptor.

### Syscall binding

This binding type can be used by systems that support multiple address spaces to provide application isolation. The syscall binding allows an application to use services provided by the kernel. A syscall binding is implemented using a client stub that performs a hardware syscall instruction `sc`, thus triggering an exception. The syscall trap handler then calls the target interface component. The application can pass up to seven arguments in registers (`r4` through `r10`) to the target. The remaining arguments, if any, must be passed in shared memory or on the user stack.

An optimization of the syscall binding can exploit the System V ABI specification calling conventions [35]. Registers (`r1`, `r14` to `r31`) are non volatile between method calls and it is not necessary to save them in the calling stub. Other registers (`r0`, `r3` to `r13`) are lost during method calls, and it is not necessary to save them either. Obviously this optimisation assumes that the ABI call conventions are obeyed. This optimization can save about 70 cycles per syscall.

### Upcall and Signal binding

The upcall and signal bindings allow the kernel to interact with an application. A signal binding is used to propagate an exception to the currently running application, while an upcall binding is used to propagate an exception to an application running in a different address space than the current one. Upcall and signal bindings are very efficient because they merely invoke a dedicated handler in the application. The binding first updates the instruction and stack pointers, and then invokes the handler in the application using the special instruction `rfi`. The exception context is also propagated. This handler then calls the target component interface, which is designated by its memory address stored in the `r3` register.

Because the exception context is propagated, the upcall binding is not completely secure: an upcalled component may never return, thus monopolizing the processor. Several standard solutions can be used to build a secure upcall binding, for instance activating a timeout (unmasked prior to switching control to the upcalled address space) or using shared memory and a yield mechanism to implement a software interrupt.



## Synchronous LRPC binding

An LRPC binding implements a simple synchronous interaction. It uses the syscall and upcall bindings. The syscall binding stub directly calls the upcall stub which calls the target application component interface.

## Remote RPC binding

A remote binding implements a simple remote operation invocation protocol, which provides transparent access to components on a remote host. The binding directly builds and sends Ethernet packets, using the network protocol components. Although the binding is designed to work between kernels, it can support interaction between remote applications when combined with the syscall and upcall bindings.

## 5 Evaluation

In this section, we describe several experiments in assembling different operating system kernels using THINK. We have implemented a minimal extensible distributed micro-kernel, a dedicated kernel for an active router, one for a Java virtual machine, and another for running a DOOM game on a bare machine.

All measurements given in this paper are performed on Apple Power Macintoshes containing a PowerPC G4 running at 500Mhz (except for the PlanP experiment, which has been done on a PowerPC G4 at 350Mhz), with 1MB external cache and 128MB memory. Network cards used in our benchmarks are Asanté Fast PCI 100Mbps cards based on a Digital 21143 Tulip chip.

### 5.1 An extensible, distributed micro-kernel

We have built a minimal micro-kernel which uses L4 address space, and thread models. Instead of L4 IPC, we used KORTX LRPC binding. The resulting kernel size is about 16KB, which can be compared with a 10KB to 15KB kernel size for L4 (note that L4 has been directly hand-coded in assembly language). Figure 7 depicts the component graph associated with this minimal micro-kernel. The figure shows the relationships between resources and resource managers, interfaces exported by components, as well as language bindings and

local bindings used for combining the different components into a working kernel.

Table 3 summarizes the performance of synchronous bindings provided by the KORTX library. Each call has a single argument, the `this` pointer, and returns an integer. An interaction via a local binding takes 6 cycles. This shows that a basic interaction between THINK components does not incur a significant penalty. The KORTX syscall binding takes 150 cycles, which can be reduced to only 81 cycles when applying the optimisation described in section 4.5. By comparison, the Linux 2.4 syscall for implementing the `getpid` syscall takes 217 cycles.

Interaction	instructions	time( $\mu$ s)	cycles
local	6	0.016	8
syscall	115	0.300	150
optimized syscall	50	0.162	81
signal	35	0.128	64
upcall	107	0.346	173
LRPC	217	0.630	315
optimized LRPC	152	0.490	245

Table 3: Performance of KORTX bindings

Adding a dynamic loader component to this small micro-kernel yields a dynamically extensible kernel, although one without protection against faulty components and possible disruptions caused by the introduction of new components. The size of this extensible kernel is about 160KB with all components, including drivers and managers needed for loading code from a disk.

By adding remote RPC components to the extensible kernel, we obtain a minimal distributed system kernel, which can call previously exported resources located on remote hosts.

Table 4 shows the costs of interaction through our remote RPC binding. The table gives the time of completion of an operation invocation on a remote component, with null argument and an integer result. The measurements were taken with an Ethernet network at 10 Mbps and at 100 Mbps. A standard reference for low-latency RPC communication on a high speed network is the work done by Thekkath et al. [32]. Compared to the 25Mhz processor used in their test, a back of the envelope computation<sup>5</sup> would indicate that our results are on a par with this earlier work<sup>6</sup>. Furthermore, the KOR-

<sup>5</sup> $(500/25) * (11.3 + 4) = 306$  microsecond at 10Mbps to compare with 296 microsecond found in [32].

<sup>6</sup>Especially since the breakdown of the costs is consistent with

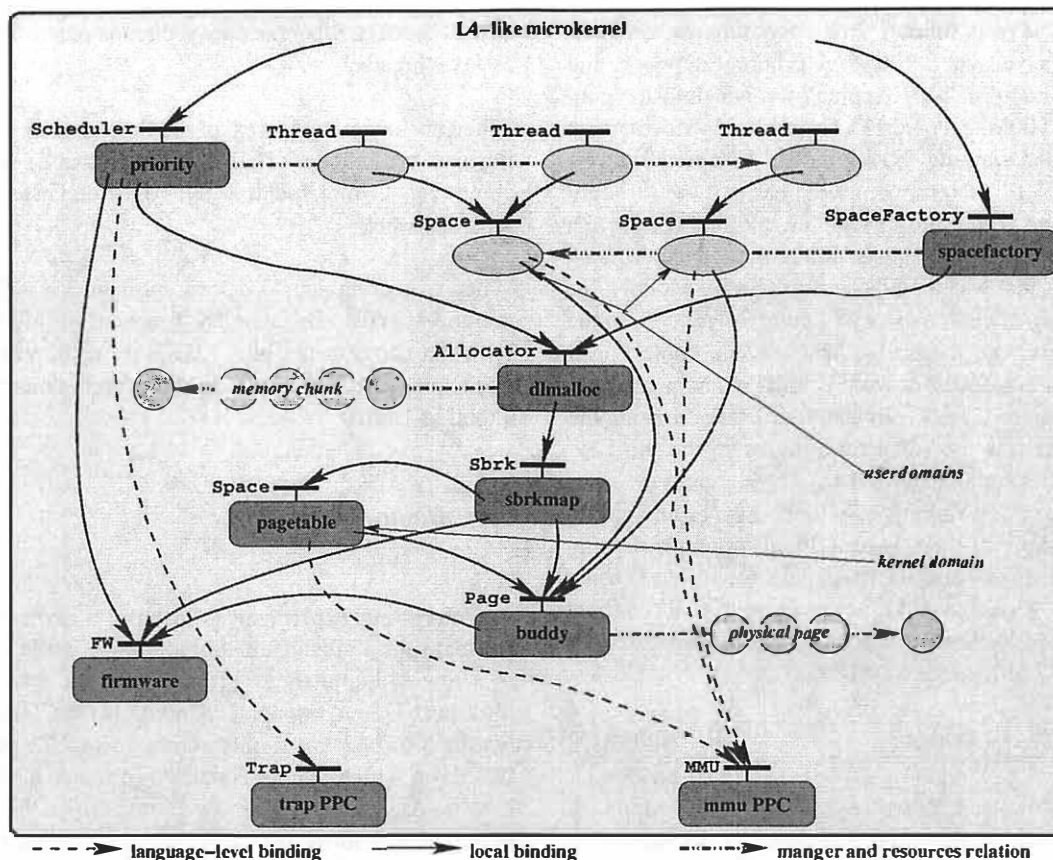


Figure 7: An example kernel configuration graph

TEX remote RPC binding can be compared with various ORBs such as Java RMI. Here, the 40 microsecond synchronous interaction performance (even adding the costs of syscalls at both sites) should be compared with the typical 1 millisecond cost of a similar synchronous interaction.

Network type	total	Time ( $\mu$ s)		
		network link	driver	marshall. +null call
10baseT	180	164.7 (91.5%)	11.3 (6.3%)	4 (2.2%)
100baseT	40	24.7 (61.7%)	11.3 (28.3%)	4 (10%)

Table 4: Performance of synchronous remote binding

These figures tend to validate the fact that the THINK framework does not preclude efficiency and can be used to build flexible, yet efficient kernels.

those reported in [32].

## 5.2 PlanP

PlanP [33] is a language for programming active network routers and bridges, which has been initially prototyped as an in-kernel Solaris module, and later ported to the Linux operating system (also as an in-kernel module). PlanP permits protocols to be expressed concisely in a high-level language, yet be implemented efficiently using a JIT compiler. While PlanP programs are somewhat slower than comparable hand-coded C implementations, a network intensive program such as an Ethernet learning bridge has the same bandwidth in PlanP as in the equivalent C program. This suggests that the Solaris and Linux kernels must be performance bottlenecks.<sup>7</sup>

To show that we can get rid of this bottleneck in a THINK system, we took as an example a learning bridge protocol *plearn*, programmed in PlanP, and we measured throughput on Solaris, Linux and a dedicated kernel built with KORTX. The configurations used in our four ex-

<sup>7</sup>PlanP runs in the kernel in supervisor mode; there is no copy of packets due to crossing kernel/user domain boundaries.

periments were as follows. In all experiments, the hosts were connected via a 100Mbps Ethernet network, and the two client hosts were Apple Power Macintoshes containing a 500Mhz PowerPC G4 with 256Mb of main memory and running the Linux 2.2.18 operating system. In the first experiment we measured the throughput obtained with a null bridge, i.e. a direct connection between the two client hosts. In the second experiment, the bridge host was a 167Mhz Sun Ultra I Model 170s with 128Mb of main memory running Solaris 5.5. In the third experiment, the bridge host was an Apple Power Macintosh G4 350Mhz with 128Mb of main memory running Linux 2.2.18. In the fourth experiment, the bridge host was the same machine as in the third experiment but running KORTEx. Throughput was measured using `ttcp` running on client hosts. Table 5 shows the throughput of the `PlanP` program running on Solaris, Linux and KORTEx. As we can see, using the KORTEx dedicated kernel increased the throughput more than 30% compared to Linux (from 65.5Mbps for Linux to 87.6Mbps for KORTEx).

bridge	throughput
none	91.6Mbps
PlanP/Solaris, Sparc 166Mhz	42.0Mbps
PlanP/Linux, PowerPC 350Mhz	65.5Mbps
PlanP/KORTEx, PowerPC 350Mhz	87.6Mbps

Table 5: Performance of the THINK implementation versus Solaris and Linux implementation

### 5.3 Kaffe

Kaffe is a complete, fully compliant open source Java environment. The Kaffe virtual machine was designed with portability and scalability in mind. It requires threading, memory management, native method interfacing and native system calls. Kaffe was ported to a dedicated THINK kernel by mapping all system dependencies to KORTEx components. For example, exception management makes direct use of the exceptions HAL component, whereas preemptive threads have been implemented on both the priority-based scheduler, which provides a native thread like semantics, and the cooperative scheduler which provides a Java thread like semantics. Thanks to our binding and component framework, making this change requires no modification in the threading code. Table 6 compares the performance of Kaffe when running on Linux and when running on our dedicated kernel. As we can see, exception management is better on the dedicated kernel due to the reduced

cost of signals, whereas native threads perform as well as Java threads.

When porting a JVM, most of the time is spent in adapting native methods. Thanks to the reuse of KORTEx components, implementing the Kaffe dedicated kernel took one week.

When executing standard Java applications with small memory needs, the memory footprint is 125KB for KORTEx components, plus 475KB for Kaffe virtual machine, plus 1MB for bytecode and dynamic memory, for a total of 1.6MB.

### 5.4 Doom

An interesting experiment is to build a dedicated kernel that runs a video game (simulating e.g. the situation in a low-end appliance). To this end, we have ported the Linux Doom, version LxDoom [16], to THINK, using the KORTEx flat memory component. The port took two days, which mainly consisted in understanding the graphic driver. The memory footprint for this kernel is only 95KB for KORTEx components, 900KB for the Doom engine and 5MB for the game scenario (the WAD file).

The THINK implementation is between 3% and 6% faster than the same engine directly drawing on the frame-buffer and running on Linux starting in single user mode, as shown in table 7. Since there are no system calls during the test, and the game performs only computation and memory copy, the difference is due to residual daemon activity in Linux and to the use of the flat memory which avoids the use of the MMU. To pinpoint the cost of the latter, we have built the same application by simply switching to the use of the KORTEx paged memory management component. As we can see, the use of the MMU adds about 2% on the global execution time. While the performance benefits are barely significant in this particular case, this scenario illustrates the potential benefits of the THINK approach in rapidly building optimized, dedicated operating system kernels.

	external resolution		
	320x200	640x480	1024x768
KORTEx(flat)	1955	491	177
KORTEx(MMU)	1914	485	171
Linux	1894	483	167

Table 7: Doom frames per second

Benchmark	Kaffe/Linux (java-thread)	Kaffe/KORTEX (java-thread)	Kaffe/KORTEX (native-thread)
synchronized(o) {}	0,527 $\mu$ s	0,363 $\mu$ s	0,363 $\mu$ s
try {} catch(...) {}	1,790 $\mu$ s	1,585 $\mu$ s	1,594 $\mu$ s
try {null.x()} catch(...) {}	12,031 $\mu$ s	5,094 $\mu$ s	5,059 $\mu$ s
try {throw} catch(...) {}	3,441 $\mu$ s	2,448 $\mu$ s	2,434 $\mu$ s
Thread.yield()	6,960 $\mu$ s	6,042 $\mu$ s	6,258 $\mu$ s

Table 6: Evaluation of the Kaffe dedicated THINK kernel

## 6 Assessment and Future Work

We have presented a software framework for building flexible operating system kernels from fine-grained components and its associated tools, including a library of commonly used kernel components. We have evaluated our approach on a PowerPC architecture by implementing components providing services functionally similar to those implemented in the L4 kernel, and by assembling specific kernels for several applications: an active network router, a Java virtual machine, and a Doom game. The micro-benchmarks (e.g. context switching costs and binding costs) of our component-based micro-kernel show a level of performance that indicates that, thanks to our flexible binding model, building an operating system kernel out of components need not suffer from performance penalties. The application benchmarks for our example dedicated kernels show improved performances compared to monolithic kernels, together with smaller footprints. We have also found that developing specific operating system kernels can be done reasonably fast, thanks to our framework, component library, and tools, although our evidence in this area remains purely anecdotal.

This encourages us to pursue our investigations with THINK. In particular, the following seem worth pursuing:

- Investigating reconfiguration functions to support run-time changes in bindings and components at different levels in a kernel while maintaining the overall integrity of the system.
- Investigating program specialisation techniques to further improve performance following examples of Ensemble [15] and Tempo [20].
- Developing other HAL components, in particular for low-end appliances (e.g. PDAs), as well as ARM-based and Intel-based machines.

- Developing a real-time OS component library and exploiting it for the construction of an operating system kernel dedicated to the execution of synchronous programming languages such as Esterel or Lustre.
- Exploiting existing OS libraries, such as OSKit, and their tools, to enhance the KORTEX library and provide a more complete development environment.

### Availability

The KORTEX source code is available free of charge for research purposes from the first two authors.

### Acknowledgments

Many thanks to our shepherd, F. Bellosa, and to anonymous reviewers for their comments. S. Krakowiak helped us tremendously improve on an earlier version of this paper.

This work has been partially supported by the French RNRT Project Phenix.

### References

- [1] ITU-T Recommendation X.903 | ISO/IEC International Standard 10746-3. *ODP Reference Model: Architecture*. ITU-T | ISO/IEC, 1995.
- [2] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for unix development. In *Proceedings of the USENIX Summer 1986 Conference*, pages 93–112, 1986.
- [3] B.N. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Ex-

- tensibility, safety and performance in the SPIN operating system. In SOSP'95 [28], pages 267–283.
- [4] B. Dumant, F. Horn, F. Dang Tran, and J.B. Stefani. Jonathan: an Open Distributed Processing Environment in Java. In *Middleware'98* [17].
  - [5] eCos. <http://sources.redhat.com/ecos/>.
  - [6] D.R. Engler, M.F. Kaashock, and J.W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP'95 [28], pages 251–266.
  - [7] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In SOSP'97 [29], pages 38–51.
  - [8] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 267–282, Monterey, CA, USA, June 1999.
  - [9] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of  $\mu$ -Kernel-based systems. In SOSP'97 [29], pages 66–77.
  - [10] R. Hayton, M. Bursell, D. Donaldson, and A. Herbert. Mobile Java Objects. In *Middleware'98* [17].
  - [11] N. C. Hutchinson and L. L. Peterson. The  $\alpha$ -Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.
  - [12] F. Kon, R.H. Campbell, M.D. Mickunas, K. Nahrstedt, and F.J. Ballesteros. 2K: a distributed operating system for heterogeneous environments. Technical Report UIUCDCS-R-99-2132, University of Illinois, Dec 1999.
  - [13] I. Leslie, D. McAulcy, R. Black, Timothy Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE JSAC*, 14(7), 1997.
  - [14] J. Liedtke. On  $\mu$ -kernel construction. In SOSP'95 [28], pages 237–250.
  - [15] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In SOSP'99 [30].
  - [16] Lxdoom. <http://prboom.sourceforge.net/>.
  - [17] *Proceedings of the 1998 IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Lake District, UK, September 1998.
  - [18] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashock. The Click Modular Router. In SOSP'99 [30].
  - [19] D. Mosberger and L. Peterson. Making Paths Explicit in the Scout Operating System. In OSDI'1996 [21].
  - [20] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Gocl. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.
  - [21] *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.
  - [22] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandekar, A. Purtell, and J. Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, 19(3):473–487, 2001.
  - [23] QNX Software Systems Ltd, Kanata, Ontario, Canada. <http://www.qnx.com>.
  - [24] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, California, October 2000.
  - [25] V. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, April 1992.
  - [26] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In OSDI'1996 [21], pages 213–227.
  - [27] M. Shapiro. A binding protocol for distributed shared objects. In *Proceedings of the 14th International Conference on Distributed Computer Systems (ICDCS)*, Poznan, Poland, June 1994.
  - [28] *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, December 1995.
  - [29] *Proceedings of the 1997 ACM Symposium on Operating Systems Principles*, St-Malo, France, October 1997.
  - [30] *Proceedings of the 1999 ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.
  - [31] S-T. Tan, D. K. Raila, and R. H. Campbell. A Case for Nano-Kernels. Technical report, University of Illinois at Urbana-Champaign, Department of Computer Science, August 1995.
  - [32] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, 1993.
  - [33] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, October 1998.
  - [34] Wind River Systems Inc, Alameda, CA, USA. <http://www.windriver.com>.
  - [35] S. Zucker and K. Karhi. System V Application Binary Interface: PowerPC Processor Supplement. Technical report, SunSoft, IBM, 1995.



# Ninja: A Framework for Network Services

J. Robert von Behren, Eric A. Brewer, Nikita Borisov, Michael Chen, Matt Welsh  
Josh MacDonald, Jeremy Lau, Steve Gribble\* and David Culler  
*University of California at Berkeley*

*Ninja is a new framework that makes it easy to create robust scalable Internet services. We introduce a new programming model based on the natural parallelism of large-scale services, and show how to implement the model. The first key aspect of the model is intelligent connection management, which enables high availability, load balancing, graceful degradation and online evolution. The second key aspect is support for shared persistent state that is automatically partitioned for scalability and replicated for fault tolerance. We discuss two versions of shared state, a cluster-based hash table with transparent replication and novel features that reduce lock contention, and a cluster-based file system that provides local transactions and cluster-wide namespaces and replication. Using several applications we show that the framework enables the creation of scalable, highly available services with persistent data, with very little application code — as little as one-tenth the code size of comparable stand-alone applications.*

## 1 Introduction

The Ninja Project is focused on Internet infrastructure and the need for a better way to create, maintain and operate robust giant-scale distributed systems. Although the overall project [GW<sup>+</sup>01] addresses wide-area systems, in this paper we study building robust large-scale centralized network services. Thus we focus on clusters within a single administrative domain that act as a centralized server for many users and potentially many services. The primary goal is to deal in full with the word “robust”, which includes basic problems of scalability, availability, fault tolerance, and persistence.

Network services include almost all aspects of large web sites, including many non-HTTP services, such as instant-messaging, e-mail and the central-server aspects of peer-to-peer file sharing. These services have a form of *natural parallelism* that derives from supporting millions of independent users; we thus define scalability, concurrency and high availability in terms of users or requests. The basic unit of work is thus a query or connection (depending on the service) from a specific user.

We believe that the framework presented here is the right way to build these services: both that the programming model is the right way to think about the service, and that the mechanisms we use greatly simplify service authoring. In some sense, this framework is our fourth

version over a period of five years (starting with [FGCB97]) and therefore represents considerable refinement of both the model and mechanisms. Unfortunately, it is nearly impossible to prove that a framework is “right” — instead we focus on describing the principles and invariants provided by the framework and why they simplify service authoring, and we examine the code size of several representative services and show that they are remarkably small given that they are scalable, highly available and persistent in the presence of faults.

We explicitly do not look at those parts of a site built on top of a database management system (DBMS) for several reasons. First, there is much work in industry on this topic and several products that work well. Second, our work is complementary to database research and would be easy to integrate with a DBMS by using Ninja as an “application server”. Third, we tend to focus on high availability, rather than transactions, and support a wider range of semantics than ACID [GR97]. However, we do look at persistence, replication, atomicity, and consistency, and many things done with a database are perhaps better done directly in Ninja (see Section 6).

The requirements for network services are very demanding. By “robust” we mean all of the following:

**Scalability:** the ability to support 100M users.

**High Availability:** the ability to answer queries nearly all of the time. Ninja services should be able to reach 4 or 5 nines, that is, the probability of answering a query should be above 0.9999 (when desired). High availability means that most queries succeed and that if a query fails, retrying it has a high probability of success: ideally, retries should be independent trials. This differs from the harder goal of “fault tolerance” in which a query must complete correctly without a client-visible retry.

**Persistent Data:** Like high availability, this is a specific form of fault tolerance: that data survives faults. This requires replication, and much of the framework will deal with automating replication for availability and persistence. There is often, but not always, an implied sub-goal of *consistency* for the data. We support a range of performance and consistency tradeoffs, with the default being linearizability [HW87].

**Graceful Degradation:** We cannot assume that there will be sufficient resources to always handle the offered load. Instead, we aim for graceful

\*: Now at the University of Washington  
This work was supported in part by DARPA #DABT 63-98-C-0038.



degradation through admission control and prioritization of requests. We aim to achieve the maximum throughput even when overloaded.

**Online Evolution:** A variation of high availability, online evolution is the ability to upgrade the service in place without significant downtime. In most cases, we can upgrade a service without downtime.

One primary goal is to make achieving these properties *easy* for service authors. We have developed several example applications that exhibit robustness; we judge ease of authorship primarily by code size. To achieve ease of authorship, we follow employ three principles:

**Exploiting Clusters:** In a data-center environment, we can make many assumptions that are not true in general for distributed systems. These include a reliable source of power, temperature control, physical security, 24-hour monitoring, and a partition-free internal network.

**Programming Model:** We believe that a fully general programming model makes it impossible to provide robust services. Instead, we use namespaces and narrow interfaces to control the sharing, replication, and persistence of data, which means that we do not have to provide these properties for all data at all times. Second, we forsake general multi-threaded concurrency for a specific style that matches the natural parallelism. We thus focus on inter-task parallelism rather than intra-task parallelism, although we support asynchronous I/O. We show that this model is sufficiently expressive to write a wide variety of services.

**Hide Complexity:** We share with DBMS research the goal of hiding the complex details of replication, persistence, load balancing and fault tolerance from applications. However, we do so through the use of reusable data structures and libraries rather than via an abstract data model and a declarative language (SQL). Both approaches enable strong properties with relatively little application code, but our approach fits more naturally with applications written in imperative languages such as C or Java.

We define the programming model in Section 2, and our key mechanisms in Section 3. Section 4 describes the applications and Section 5 presents their evaluation. Section 6 discusses our principles and related and future work, and Section 7 provides a summary.

## 2 Programming Model

The goal of the programming model is to simplify the creation of complex network services; such services must map naturally onto the programming model. Second, the model must enable an underlying implementa-

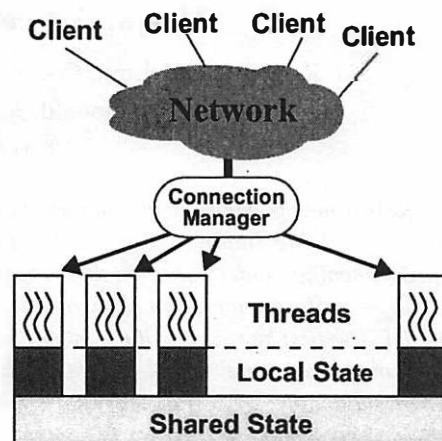


Figure 1: The Programming Model

A node consists of threads, local state and shared state. Nodes use the same "program" (code base), and receive connections from the Connection Manager in the style of data parallelism.

tion that hides the details of fault and load management, scalability, high availability, and online evolution.

Given the natural parallelism described above, we choose a model based on request parallelism, in which we aim to partition users' request streams across nodes. For ease of authoring, we would like to have a single program that is automatically spread across the cluster. Thus we choose to base our model on the *single-program-multiple-data* (SPMD) model commonly used in parallel computing [DGNP88]. We make two extensions to the SPMD model: support for shared state<sup>1</sup> and management of connections to the outside world. We refer to this new model as the *single-program-multiple-connection* (SPMC) model, as shown in Figure 1. We also assume many threads per node, which differs from SPMD in practice, but not in definition. There are expected to be many connections per node, and there may be more or fewer threads than connections. One big practical difference of course is that we seek to achieve high availability and tolerance for partial failures, whereas SPMD was developed in the context of the all-or-nothing fault models of parallel machines.

By "shared state" we mean that the threads and connections active on any subset of nodes may share global namespaces that support linearizable updates (i.e. strongly consistent, see [HW87]) to network-accessible storage in a uniform manner across the cluster. This notion does not require shared memory, as assumed in the original SPMD work; instead we provide multiple global namespaces accessed via method calls rather than load/store instructions. This narrow interface to shared

1: Although many SPMD systems had at least a shared namespace (e.g. CM-5 [HT93] and T3E [Sco96]), support was inconsistent and we thus treat this as an extension.

state simplifies consistency, replication and persistence.

Conversely, we define non-shared state as “local state”, which includes local memory and files. Local state is simpler and faster to access than shared state, and is useful for session and thread state (e.g., stacks), caching shared state, and temporary files.

*Invariant: Shared state is strongly consistent across all nodes used by a service.*

Globally named shared state implies that any connection can be serviced from any node, which is a tremendous simplification. Shared state enables simple construction of groupware services, communication services (e.g. chat), and information dispersion (e.g. stock quotes); it also simplifies service management. For example, it is easy to count users, put them in (shared) queues, and track aggregate statistics about the service.

Shared state can also be highly available and persistent (up to some configurable number of faults):

*Invariant: Shared state is highly available and persistent (when desired).*

High availability requires automatic replication, while durability requires management of replicas and disk writes. The power and simplicity of the SPMC model come from the automatic management of consistent, highly available, persistent shared state. Finally, we allow services to relax these invariants for better performance (Sections 3.4.1 and 5.5).

We support multiple independent namespaces for shared data. This provides both encapsulation and logical isolation of different services and system components, thus providing a simple form of security: services cannot read or write the shared state of other services or of shared system components. Additionally, support for many namespaces enables fine-grain control over consistency, availability (replication), and persistence, all of which are attributes of a namespace.

We have found two kinds of shared state to be particularly useful: shared data structures and shared files:

**Shared Data Structures:** These have the same interface as normal data structures and are therefore very easy to use. We provide hash tables and B-trees. Hash tables are sufficient to support other models including tuple spaces and shared arrays. We also provide extensible atomic operations that enable programmers to create high-concurrency sharing primitives, such as compare-and-swap.

**Shared Files:** Although we can store large items persistently in the hash table, we found that file usage is sufficiently different and common to support directly. Some of the key differences include larger objects, larger working set sizes,

lower expectation of being in memory, and the need for data streaming over the network.

The second extension in SPMC is explicit support for connection management. The SPMD model does not define how outside I/O interacts with the nodes, except for possibly spreading files across the nodes. For network services the problem is much more dynamic: some state is long lived, and we must isolate down nodes from the clients to provide high availability.

*Invariant: New or retried connections arrive at “up” nodes.*

Note that we do not promise that connections do not go down: existing connections are lost when a node goes down. Although possible in theory, moving active connections when the server side dies is not practical. For example, every potentially client visible state change must be durable, which requires tracking those changes to either a replica or a persistent store, as they occur. Instead, we promise that retried connections are not affected by the failure. Using shared state, it is possible to keep session state across this transition as needed, which is simpler and much more tractable than tracking all client-visible state automatically.

To enable more intelligent connection management, we add one key idea to the programming model: connections are partitioned into application-defined classes, which we call *partitions*. By default a service has only one class, in which case all connections are treated the same, but in practice explicit partitioning gives the author more control over the service. In particular, a partition is the:

**Unit of Affinity:** Connections in the same partition go to the same node(s), which enables cache affinity (similar to LARD [PAB+98]), and reduces communication for users within the partition. For example, if all users in a chat room are in the same partition, then the group state resides on that node.<sup>2</sup>

**Unit of Priority:** Partitions allow the author to control graceful degradation and quality of service. In particular, we can support application-defined admission control, by dropping connections in low-priority partitions first. The same idea enables differentiated quality of service by partition: we can support different densities of users/node for high- and low-priority partitions. For example, high-paying stock traders might have less congestion and thus faster trades, especially during overload.

---

<sup>2</sup> Some communication is still required if the chat state is replicated, but typically chat rooms are neither persistent nor highly available; the application code would be almost identical regardless.

**Unit of Migration:** Under a load imbalance or a fault, it may be necessary to migrate users' state to a new node. Partitions are the unit of migration for fault recovery and load balancing. This ability also simplifies online evolution, as we can do a rolling upgrade by partition.

In general, explicit partitions are powerful because we get simple application-level guidance on how to group connections. We can then use these groups to provide fine-grain control over replication, cache affinity, quality of service, graceful degradation and online evolution. Note that we partition connections and not users; we can use them to partition users or we can have the same user in different partitions simultaneously depending on the task. Finally, service authors can ignore partitions if they need only even load balancing.

### 3 Mechanisms

In this section we examine the four key building blocks that we use to achieve the SPMC model.

#### 3.1 Clone Groups

The first mechanism is to virtualize the SPMC model: instead of each service running on a whole cluster, we instead run services on *clone groups*, which are a set of *clones* with common code and state. A clone is a virtual node that we map onto a real node dynamically; we refer to them as clones because they share the same code base (the "single program") and shared state. Thus when we discuss shared state or namespaces, it is always for a specific clone group. Similarly, connections are managed across clone groups, not the cluster.

*Principle: Clone groups provide each service with a virtual cluster.*

Clone groups typically map onto a subset of the real nodes, and may vary in size depending on load. More than one clone group may map onto a node, in which case they are isolated in terms of state and namespaces, but not in performance. However, the connection manager described below can maintain even load balancing even if clones have uneven throughput due to differences in hardware, work per connection, or interference from other groups.

Clone groups provide several useful mechanisms to the programmer, including membership, broadcast and barrier synchronization. Changes in membership lead to notification of all clones via birth and death events. Membership is approximate and eventually consistent, which has proven sufficient in practice. For example, we use death notification to instigate recovery within the shared data structures.

Broadcast is mostly useful for notification, since there is a better mechanism for sharing state. Barriers

could be implemented on top of shared state, but are actually done via message passing (i.e. events) because of the need to integrate dynamic membership information. A barrier is considered done when all live nodes reach the barrier, so death events may complete a barrier. As with SPMD, barriers are used to ensure that all clones are in the same stage; the biggest use seems to be to denote the completion of an initialization phase.

An overall manager, called the "shogun", dynamically modifies the size of each service's virtual cluster based on utilization. Remarkably, most services don't care about the size of the cluster, since the shared state is managed across the transition automatically, and no connections are lost during the transition (see Section 5.4). A service can track changes using the birth/death events when needed.

Typically, replication uses subsets of a clone group of storage nodes. A *replica group* is thus a subset of a clone group that handles replication for part of the shared data, so that we can decouple the degree of replication from the number of clones. Replicas use the clone-group mechanisms to handle replica membership and synchronization. We use many small replica groups in one storage clone group, with overlapping membership. For example, with 2-way replication, a replica group is a two-node subset of a larger storage clone group. The use of lots of small groups reduces the recovery latency per group, and enables incremental recovery, where each small group is one step. By design, the groups are small enough that we can just copy the whole contents of another replica atomically, without too much concern for the fact that we prevent updates to that group (only) during the copy.

#### 3.2 Single-node Run-Time System: SEDA

An important aspect of building scalable services is to support very high concurrency and to avoid overcommitment of server resources. Building highly concurrent systems is inherently difficult: structuring code to achieve high throughput is not well-supported by existing programming models, and traditional concurrency mechanisms, particularly threads, make it difficult for applications to exercise control over their resource usage.

Ninja makes use of a concurrency design called SEDA, or *staged event-driven architecture*. Services are structured as a set of *stages* connected by explicit *event queues*. This design permits each stage to be individually conditioned to load (e.g., by performing thresholding on its incoming event queue), and facilitates modular application construction. SEDA, covered in detail in [WCB01], enables not only very high concurrency, but also graceful degradation through resource management and adaptive load shedding.

For the purposes of this paper, SEDA provides two key capabilities: support for more connections/node and thus better overall performance, and detection of overload at the node level, which we need to provide graceful degradation for the overall service.

The scalability limits of threads are well-known and have been studied in several contexts, including Internet services [PDZ99] and parallel computing [RV89]. Generally, as the number of threads grows, OS overhead (scheduling and aggregate memory footprint) increases, which leads to a decrease in overall performance. Direct use of threads presents several other correctness and tuning challenges, including race conditions and lock contention.

*Principle: Concurrency is implicit in the programming model; threads are managed by the runtime system.*

Since we must avoid excess threads to achieve graceful degradation, we simply prevent services from creating threads directly. Instead, services only define *what could be concurrent*, via (explicitly) concurrent stages. Conceptually, each stage has a dedicated but bounded thread pool, but the allocation and scheduling of threads is handled by SEDA. Thus the system as a whole is event-driven, but stages may block internally (for example, by invoking a library routine or blocking I/O call), and use multiple threads for concurrency. The size of the stage's thread pool must be balanced between obtaining sufficient concurrency and limiting the total number of threads; SEDA uses a feedback loop to manage thread pools automatically. The particular policies are beyond the scope of this paper, as they only effect the node performance. Roughly, allocation is based on effective use of threads (non idle) and priorities, while scheduling is based on queue size and tries to batch tasks for better locality and amortization (as in [Lar00]).

Internal framework modules, such as the shared-state mechanisms in Section 3.4, also use stages and avoid explicit thread creation. The internal modules are often written in the event-driven style, common for high-performance servers [PDZ99], which we enable by providing non-blocking interfaces for all network and disk activity, and for the shared data structures.

*Invariant: Overload detection is automatic; services are notified when they are overloaded.*

A key property of queues is that it becomes possible to implement backpressure by thresholding the event queue for a stage. We use this to detect overloaded stages and thus to initiate *overload mode* and graceful degradation. With only the implicit queues of blocked threads, it is difficult to detect overload until too late.

Thus our single-node runtime system provides two

key capabilities. First, it provides control over thread allocation and scheduling, which enables either thread-based or event-driven programming and ensures thread limits consistent with the operating range of the node. Second, it provides backpressure via explicit queues that enables the detection of overload and thus graceful degradation, which is shown in Section 5.3.

### 3.3 Connection Manager (CM)

The Connection Manager (CM) is responsible for all external names. It dynamically maps external names to clone groups and connections to an external name to a specific clone. It must hide failed nodes and balance load across the clone group. Although it appears in the figure as a single point of failure, it is actually a pair of “layer 7” switches [Fou01] that provide automatic failover for each other. Based on ethernet switch reliability, we estimate the uptime of these switches at about  $1 \cdot 10^{-7}$  each (seven 9's), so the pair is extremely reliable.

As an optimization, services can define partitions that are subgroups of names (and thus connections), to provide fine-grain control over resource allocation and graceful degradation. The CM can map partitions to subsets of clones in a clone group, or in the case of admission control deny partitions altogether.

#### 3.3.1 External Names

The connection manager provides a level of indirection for external service names. The CM maps external names to clone groups, which may change dynamically, and load balances connections among the clones. In general, the CM maps external (IP, port) pairs to the set of internal pairs corresponding to the clone group. When there are multiple clones, connections are balanced across the target set based on open connections.

The CM tracks clone birth and death events in order to maintain high availability. Starting a clone is a two-step process. During initialization, Ninja allocates server sockets for a clone and starts it. It then registers the clone with the CM, which starts to forward connections to the clone. Stopping a clone is the reverse process: the CM stops forwarding connections to the clone and then removes it from the clone group. To provide higher availability, the clone may finish processing outstanding requests before it exits.

*Invariant: Ninja can remap or resize clone groups without dropping connections.*

The ability to shutdown clones gracefully makes it possible for Ninja to remap clones to nodes dynamically or to reduce the size of an underutilized clone group. The same ability enables online evolution to a new version with no downtime (shown in Section 5.4).

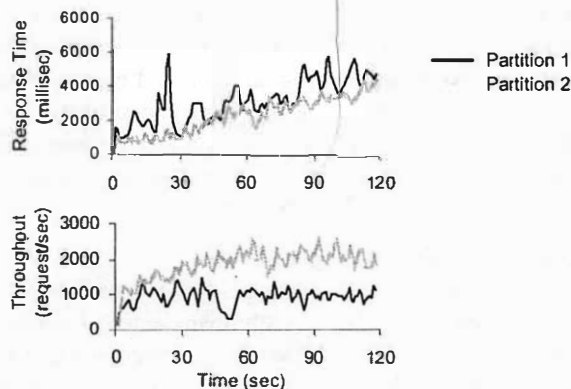


Figure 2: Delivering different Qualities of Service  
A three-node web server is divided into two partitions: Partition 1 maps onto one node, Partition 2 onto two. The CM achieves twice the asymptotic throughput and better response time for the larger partition, and both tiers show graceful degradation.

### 3.3.2 Partitions

In the current implementation, services can define partitions in two ways, either via ports or URL string matching. With ports, services map external port numbers to partitions, which are then dynamically mapped to clones. Typically, services would use one external port number per partition, although more are allowed. For HTTP requests, the service can define partitions based on URL hashing and string matching; we currently support prefix, suffix, and substring matching. This can be done at wire speed using current “layer 7” switches [Fou01]. Given these partitions, the CM will dynamically map partitions to clones.

*Principle: Partitions enable division of the working set for higher throughput.*

As in LARD [PAB+98], partitions provide better locality and cache performance as the working set is partitioned across the clone group. Without partitions, the CM spreads load evenly, which effectively replicates the working set at each clone.

Partitions are also the unit of priority which helps with tiered quality of service and graceful degradation:

*Principle: Partitions enable tiered quality of service.*

First, the CM enables differential quality of service by allocating varying resources to different partitions: partitions need not map evenly onto clones. Figure 2 shows this proactive form of uneven load balancing. Partition 1 maps to one clone, while Partition 2 maps to two: the latter has twice the asymptotic throughput and better latency, particularly as Partition 1 reaches its overload point (about time 20). The variance of the one-node partition is higher as well, due to averaging effects. Note

that both graphs show graceful degradation, with smooth asymptotes and linearly increasing latency.

Second, priorities enable more graceful degradation during overload. The CM implements an admission-control policy based on partition priorities: requests to low-priority partitions are dropped first. Individual nodes can detect overload directly (Section 3.2) and notify the CM. In overload mode, the CM drops requests by routing them to a generic “drop” clone, analogous to the use of `/dev/null` in UNIX. This is complementary with the first strategy and they may be used together. In addition to admission control, nodes may take action themselves in overload mode to reduce the average work per request. We evaluate these mechanisms in Section 5.3.

There is also a relationship between failures and overload: when a clone fails, the remaining clones typically receive increased load, which may put them into overload mode. The movement of load due to failures and the reaction to overload are independent mechanisms, but both are automated and overload mode will kick in only if needed.

Finally, it is important to realize that partitions do not effect shared state or replication. In the graph above for example, all three clones have the same program and shared state, but the CM allocates traffic unevenly by partition. This means that any clone *can* handle any partition, although they do not in normal operation. If a node fails, the remaining two nodes are given both partitions automatically, which affects the difference in quality but maintains high availability.

To summarize, the connection manager provides management of all external names, including dynamic mapping of names to physical nodes for load balancing and fault tolerance. It also implements policies based on partitions that allow a service to define relative quality of service and prioritized admission control.

### 3.4 Shared State

The fourth mechanism provides services with shared state: currently shared hash tables, B-trees, and file systems. The shared state mechanisms need to support robust applications, and therefore must be scalable, highly available, durable and consistent. By implementing these properties in the shared state mechanisms, we essentially eliminate the burden of achieving them. In particular, we hide all of the issues of atomicity, replication, consistency and recovery from service authors.

Because high availability and consistency are incompatible in the presence of partitions [FB99], we opt to use a redundant system-area network for communication within our cluster (currently gigabit ethernet with redundant switches). A partition-free network allows us to use a two-phase commit protocol (2PC) [GR97] to



ensure consistency and atomicity of updates to shared state across several nodes. The version of 2PC we use is optimized for high availability in two ways. First, if a member of the protocol dies in the second phase, the 2PC completes without it, because the replica will be able to recover a consistent image of its state from its peers later. Second, if the coordinator fails, we cannot afford to wait until it recovers to complete the protocol; instead, the replicas contact each other proactively after a timeout and commit the action if any member received a commit; otherwise, they all abort. The protocol is also available to application writers to extend the framework with additional shared state mechanisms.

In the next two sections, we examine the cluster hash table and file system in more detail. The cluster B-tree is ongoing work.

### 3.4.1 Cluster Hash Table (CHT)

Our prototypical shared data structure is the cluster hash table, which uses the traditional interface of three operations: get, put, and remove. Each operation is atomic with strong consistency (equivalent to having a single copy). The underlying data is partitioned across the cluster for scalability, and replicated for high availability (see [GBH+00] for more details). The degree of replication can be varied based on the requirements of the application, and different tables in the same service may use different replication strategies. This control enables tradeoffs among performance, fault tolerance and storage requirements, and also enables the composition of modules without name or policy collisions.

#### 3.4.1.1 Non-Blocking Synchronization

The atomic put operation on the CHT returns the old value prior to the update, in essence implementing an atomic swap. Atomic swap can be used to implement various synchronization primitives, such as locks (using test-and-set) or read-modify-write (swapping in a “locked” value first and then the updated value). Such implementations, however, can be classified as *blocking* [Her91], in that a process holding a lock may take an arbitrary time to complete. This reduces both scalability, due to lock contention, and availability, since a process may die while holding the lock.

To overcome these obstacles, we extended the hash table interface with an *apply* operation, which implements an atomic read-modify-write:

```

apply (key, update_function) {
    temp = get(key)
    put(key, update_function(temp))
    return temp
}

```

The apply operation is implemented by shipping the name of the update function to the nodes that store the

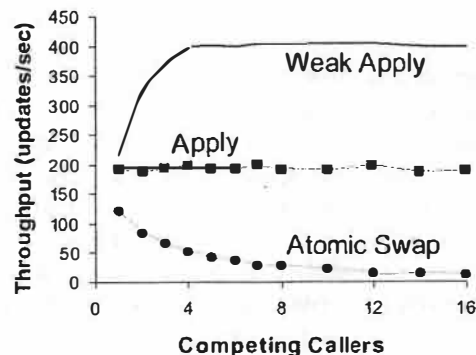


Figure 3: Optimizations for Update Contention  
This graph plots single-node throughput under heavy contention in the CHT (without replication). Atomic Swap drops off due to long lock-holding times, while Weak Apply performs twice as well as apply, by not holding locks across the 2PC round trip.

data and executing it there, analogous to function shipping in databases. We can use the name rather than the code, because of the “single program” facet of the SPMC model. Atomicity is ensured, as before, by the 2PC protocol. Read-modify-write is sufficiently general to implement a wide range of atomic primitives, such as compare-and-swap, fetch-and-add, etc. Several of these primitives, such as compare-and-swap, are *universal* [Her91], and thus can be used to build non-blocking and wait-free implementations of a data structure from a sequential one.

However, we can build non-blocking data structures directly: unlike conventional shared-memory systems, each location in a hash table stores an entire object, as opposed to just a pointer or a primitive value. This allows us to provide the update function from a sequential implementation as the argument to the apply function. The update is atomic and non-blocking.<sup>3</sup> Therefore, the operation is naturally wait-free, without the complexity or overhead usually associated with wait-free protocols.

The improved scalability of apply-based updates can be seen in Figure 3. We compare an update implemented using two atomic swaps to one implemented by an apply operation; the graph shows the aggregate throughput of several clients continuously updating the same data value. The atomic swap implementation performance quickly degrades as concurrency increases, since more time is spent trying to obtain the lock. The apply-based implementation performs better at the outset, since it requires half as many operations to complete

3: This is not strictly true (nor could it be) for an arbitrary update function; however, we assume simple non-blocking update functions, which holds in actual use. Our most complex update function appends to a list represented as an array and sometimes has to resize the array.



an update, and the aggregate throughput remains virtually flat as we scale up to 16 nodes.

The graph also shows performance of a “weak” version of the apply operation; this version has exactly the same interface, but weaker consistency semantics. Namely, it commits the update in the first phase of the 2PC; the second phase is only to let replicas know that everyone made the update (in case the coordinator fails). The update is eventually executed atomically on each node; however, cluster-wide atomicity is not achieved. In particular, updates may be executed in different orders at different nodes. These relaxed semantics allow for a significant performance improvement, since locks are held only for the duration of the local update and not across the round-trip interaction with a coordinator.

An example data structure that takes advantage of these weaker semantics is an unordered list. Insert and remove operations are commutative in unordered lists, so “weak apply” semantics are sufficient. Such lists are used by several of our applications.

### 3.4.1.2 Replication and Performance

As mentioned above, the CHT replicates data for high availability. The implementation distinguishes storage clones from the libraries that clones include to use the CHT, which decouples clone group size from which nodes actually store data. The set of storage nodes remains stable except for faults and explicit operator-controlled repartitioning. Thus, the replica groups and recovery are managed entirely within the CHT implementation, and storage clones are shared by many tables and clone groups (with namespace isolation). Replication and durability policies are table-specific, but the storage clones are not.

The degree of replication can have an impact on performance: more replicas will deliver higher read throughput, but lower throughput for updates. An extreme case of the latter effect can be observed when many updates to a single location in the hash table are attempted simultaneously: different nodes may prepare successfully for different instances of the 2PC protocol, causing all instances to fail. The chances of livelock increase with the number of conflicting updates and the degree of replication.

Most data updates are largely independent, so such conflicts do not happen frequently, but when they do occur, their impact is significant. We were forced to add an algorithm that detects livelock and serializes prepare interactions with each replica. As Figure 4 illustrates, such detection improves performance of atomic apply to be tolerable, but there is still significant degradation. If this is unsatisfactory, the application designer has the choice of reducing the amount of replication or relaxing consistency requirements by using “weak” apply, which

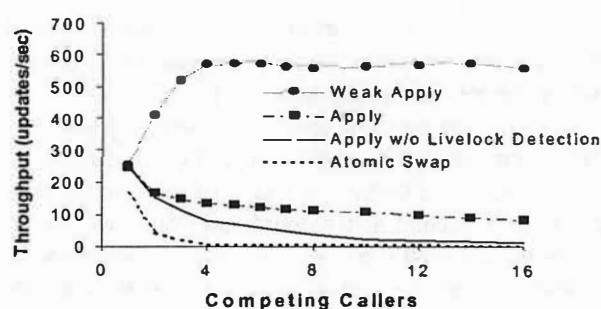


Figure 4: Update Contention with 2PC (2 replicas)  
This graph reveals the impact of livelock on updates using 2PC. Atomic Swap encounters livelock with even two competing updates. Proactive livelock detection is significant for Apply.

does not experience livelock. Another possibility, not yet implemented, would be to use exponential backoff. For comparison, we also measured the performance of atomic-swap-based updates; we found that it becomes unusable under a moderate amount of contention, despite livelock detection.

### 3.4.2 Cluster File System (CFS)

The second form of shared state is the cluster file system (CFS). In contrast to the CHT, the cluster file system manages large blobs of persistent storage that are normally on disk, and supports the streaming of data directly from disks to clients. Although it is closely related to a traditional file system, we chose not to implement the normal UNIX file system interface for several reasons:

- The traditional API limits atomicity. First, the only atomic operation is “rename” which requires copying whole files even for small updates. Second, file metadata operations are path based, which mixes path and file updates, and presents problems if the path changes during a file update. We provide first-class i-nodes, which eliminate redundant path resolutions, and provide natural support for atomic file updates, since you can name them directly.
- File consistency across multiple nodes is very limited. We desire a range of consistency and durability options, including both strong consistency across the cluster with replication, and local temporary storage.
- There is only one kind of index on files, the directory. We would like files to belong to multiple indices of different types simultaneously, including hash tables, B-trees, and version trees.
- We would like extensible metadata to simplify service-specific file operations, such as version numbers, TCP or MD5 checksums, and caching/expiration directives.

Thus, the basic strategy is to provide a “toolbox” local file system that can be reused in multiple ways to build service-specific cluster-based file systems. The toolbox deals with entirely local instances of storage, called *volumes*. We provide a simple physical global namespace using (node, volume, i-node) triplets.

A “file” consists of an i-node that has several values: typically a *segment*, which holds the data, and some metadata attributes. The metadata is extensible, which allows services to store their own metadata. A “directory” is just one kind of index on top of the i-nodes.

We provide atomic transactions, which in turn enables files to have multiple indices (or multiple parents), and simplifies renaming, deletion, and path operations. Direct exposure of i-nodes also allows database-style iteration through sets of files.

The real power of the CFS comes from the ease with which an author can create file-system like things. To build a shared file system across a clone group, the author need only define a global namespace. For example, storage for web pages need not have a directory at all, and can just use the CHT to store name→i-node mappings, thus enabling single seek access to the data segment. Or even simpler, a CFS with a fixed number of nodes can be built just by using a static hash function to map file names to nodes. Thus with little service-level code, we can achieve a variety of file systems.

Replication is completely orthogonal to the partitioning of a cluster-wide namespace and is handled quite differently than in the CHT. For a replicated CFS, the author must define the replica groups and use 2PC to update them. Since operations in the file system are atomic, the general 2PC manager can be used to build replication easily. This is intentionally quite a different policy from the CHT, in which replication was managed transparently. We take a different tack in CFS for two reasons: 1) there are a wider array of strategies for a replicated file system, making it harder to have any single one, and 2) the existence of the CHT makes it really easy to manage replica groups within a service, since it handles atomicity and recovery of this metadata automatically. This approach enables powerful service-specific CFSs (a service can have more than one) with very little application-level code. We have built three different service-specific file systems so far.

Finally, as a performance optimization, we support streaming of data segments across the cluster (versus store-and-forward copying). Any stage in the cluster may issue a stream task (acting as stream client) to another stage (the stream server), thus establishing a virtual channel within the cluster for reading or writing data. This is particularly useful for streaming data directly from the CFS out to the wide-area network, and is used for both our web and e-mail servers.

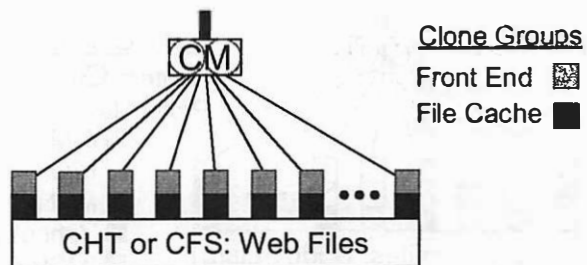


Figure 5: Ninja Web Service

Only Front Ends receive external connections, Cache nodes serve files locally or retrieve them from either the hash table or CFS (two different versions).

We have not built a generic cluster file system yet, although the one in our e-mail server is relatively general and could be packaged up for reuse by other services. We are still learning about the revised CFS API and expect to generalize support for replication and partitioning in the future, which will eliminate the small amount of service-level code required now.

## 4 Applications

In this section, we review three applications built using the Ninja framework. We then use these applications in the next section to evaluate the framework and our goals of robustness and ease of authoring. We have also built several other applications, including other web and mail servers, and a Napster-like file-sharing service.

### 4.1 Ninja Web Server

The prototypical service for Ninja is the web server, and we thus use it to evaluate all of our goals. The web server is relatively simple but achieves scalability, high availability, graceful degradation, and online evolution.

We have implemented several web server prototypes, serving both static and dynamic pages using either the hash table or the CFS for page storage. Our latest prototype builds upon the Haboob web server [WCB01] and modifies it to retrieve pages from the CHT. Haboob uses SEDA to handle a large number of simultaneous connections, making it an ideal front-end for the Ninja cluster web server. Haboob maintains an in-memory cache; we performed minor modifications to the cache miss component to fetch page data from the hash table instead of from local disk. Adding a thin wrapper to make an instance of Haboob behave as a Ninja clone allows us to create a clustered web service, with the CM directing external HTTP requests to one of the clones. Figure 5 shows the structure of the web server.

The shared persistent state maintained by the CHT allows any front-end node to answer any request; the CM masks front end failures. The replication policy used for the tables storing page data can be tuned to

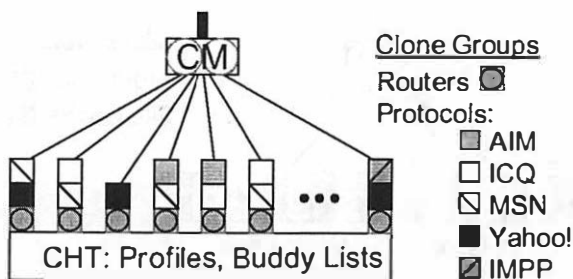


Figure 6: NinjaIM Architecture

Each protocol has a clone group, whose size depends on the traffic in that protocol. All protocols use Message Routers to communicate with each other, and all use the CHT for profiles and buddy lists.

achieve desired tradeoffs between availability and performance, and different classes of pages may be split among tables with different replication strategies. Similarly, front ends may be partitioned to provide differing quality of service or achieve better cache performance, as described in Section 3.3.2.

#### 4.2 Universal Instant Messaging Proxy

NinjaIM is an instant-messaging (IM) proxy that performs protocol translation among popular instant messaging protocols and e-mail. It currently supports AIM, ICQ, MSN, Yahoo!, and IMPP protocols. Users can use the unmodified MSN client software or our Java applet-based client to communicate with users on all five IM systems. NinjaIM forwards messages bidirectionally among the five systems, which allows all users to reach each other.

There are several challenges in implementing an IM service. First, it must scale to a huge number of connections that are mostly idle; AOL's AIM has over 90M registered users [Hu00]. Most IM systems use long-lived TCP connections for every active user. Second, it requires scalable persistent storage for user profiles and buddy lists. Third, it must be able to route messages efficiently and process buddy status updates.

Figure 6 shows the NinjaIM architecture. The Connection Manager enables NinjaIM to easily scale up the number of connections linearly with nodes, and provide high availability. The CHT is used to store user profiles and buddy lists, which allows users to connect to any node in the cluster. To provide efficient buddy status notification, both a forward buddy list and a reverse buddy list are stored. In addition, we store the node to which a user is connected, which is used to route messages between nodes. All of the shared state may be cached locally (local soft state) to improve performance. Finally, partitions are used to provide better affinity for chat sessions. For example, when a user initiates a chat session, all the parties are given the same partition number (externalized as a port number) to which to connect.

The CM maps the port number to a single node in the normal case, but need not in the presence of unusual load or faults.

#### 4.3 NinjaMail

E-mail is one of the most widely used Internet applications, with hundreds of millions of users world-wide. Moreover, many of these users are concentrated in large e-mail services. AOL currently has over 23 million e-mail accounts [Lci00], while Hotmail has over 110 million [WB01].

NinjaMail is a scalable, highly available and extensible e-mail service, built on the Ninja architecture. At NinjaMail's core is the MailStore module, a message access library that uses the CHT and CFS to store user profiles, e-mail messages, and message indices. Built on top of this are various access modules, which support interaction between users and the message store. We have fully functional modules for sending and receiving messages via SMTP, and reading messages via POP and HTML. Additionally, we have nearly completed an implementation of the IMAP protocol.

Figure 7 shows the architecture for NinjaMail. The NinjaMail modules keep all long-lived state in the CHT or the CFS. This allows the infrastructure to create and destroy clones in response to load changes or faults. NinjaMail's use of the underlying mechanisms is illuminated by examining a typical message cycle:

Message arrival (SMTP): 1) Accept a new SMTP connection from the CM, 2) check the CHT, to verify that the recipient is a valid user, 3) stream the message to the replicated file system (MailStore), and 4) use the CHT apply function to add the message to the user's message index.

Message retrieval (POP): 1) Accept a new POP connection from the CM, 2) check the user's login name and password in the CHT, 3) retrieve the user's message index, 4) stream messages from the MailStore to the user, as requested, and 5) use the CHT apply function to update the persistent copy of the message index when the user deletes messages or updates the status flags.

The cluster-based file system of the MailStore is built using the CFS to provide atomic local storage volumes, and the CHT maintains the mappings from partitions to replica groups, and from replica groups to MailStore clones. It uses the 2PC library to update the replicas. This gives us a replicated cluster file system with very little application code. There are no "directories" in the file system; the only index on files is the global hash table that maps replica groups to storage nodes.

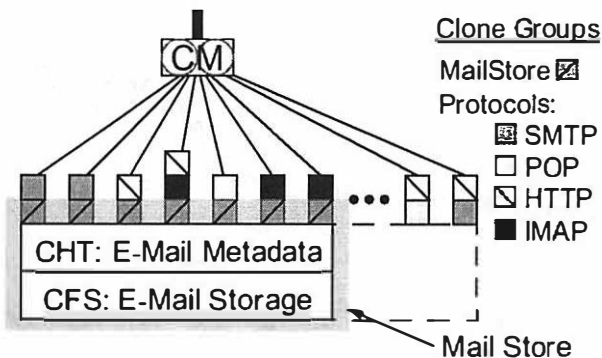


Figure 7: NinjaMail E-Mail Service

The service is divided into protocol handlers, each of which has its own clone group, and the Mail Store, which uses both the CHT and CFS to manage e-mail, user and folder information.

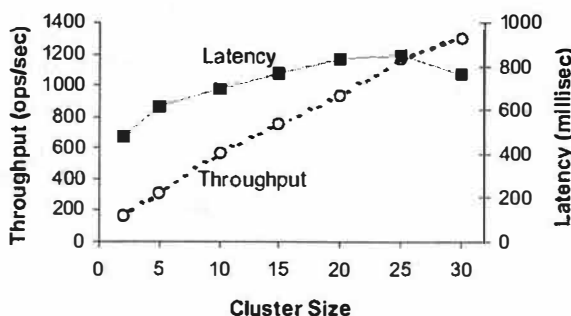


Figure 8: NinjaMail Scalability

## 5 Evaluation

In this section, we use the above applications to evaluate each of our goals: scalability, high availability, graceful degradation, online evolution, range of semantics, and ease of service authoring.

### 5.1 Scalability

For scalability, the overall goal is to support a very large number of users. For most services this corresponds directly to the number of simultaneous connections, including the web server, IM server and music server. For NinjaMail, scalability is tied more directly to messages per second. Ninja also supports linear scaling of database size, which comes directly from simple partitioning; we have built services using the CHT with more than 1TB of storage and over 100 nodes [GBH+00].

Figure 8 shows the scalability of NinjaMail in a message receipt, storage, and retrieval test. Each cluster node functions both as a front-end for SMTP and POP, and as a member of the CHT. The cluster nodes used for this experiment were 2-way SMPs with 500-Mhz processors and 512 MB of RAM, running Linux 2.4.7. Each test was performed with a user base of 1 million times the cluster size. Our test harness executes a simple loop. It first selects a random user and node, and sub-

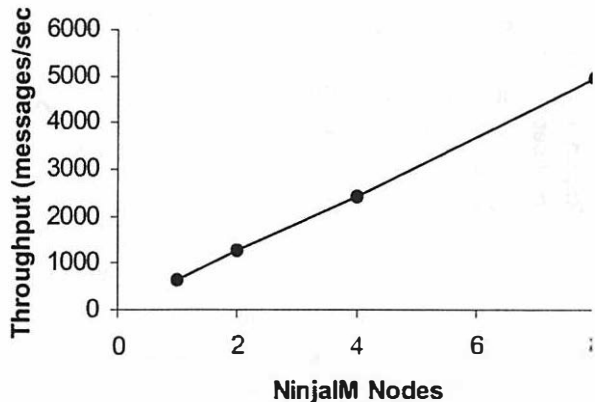


Figure 9: NinjaIM Scalability

Scalability is measured in terms of total throughput of IM messages per second. In addition to the  $n$  front-end nodes there are 2 additional nodes that store the replicated CHTs for NinjaIM.

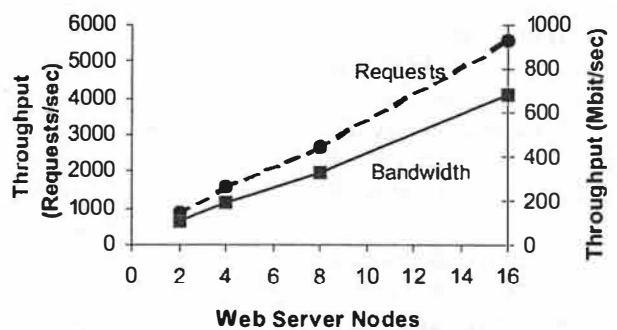


Figure 10: Web Server Scalability

mits a 4K message. Next, it selects another random user and node, and uses POP to read and delete all messages for that user. The per-node performance is excellent, at about 14 times the performance of a typical sendmail setup on the same hardware (for the message receipt portion), perhaps due the efficiency of the CHT for metadata. Extrapolating, we expect NinjaMail (as is) should be able to handle the Yahoo! mail workload, about 12 billion messages per month [Yah01], with a cluster of around 100 nodes.

Figure 9 shows the scalability for NinjaIM, measured in total throughput of IM messages. Simulated clients saturate the server using the full MSN IM protocol by sending messages every 5 seconds. Each front end node ran one message router and one MSN IM server. At the peak of 4941 messages/sec (with 8 front ends), this corresponds to almost 25,000 simultaneous extremely active clients.

Figure 10 shows the scalability of the web server under the SPECweb99 benchmark with 600MB data/node; single node performance is consistent with a solid single-node web server such as Apache [Apa01]. Note that the Ninja web server is not just a web farm, but actually reflects strongly consistent data across the clus-

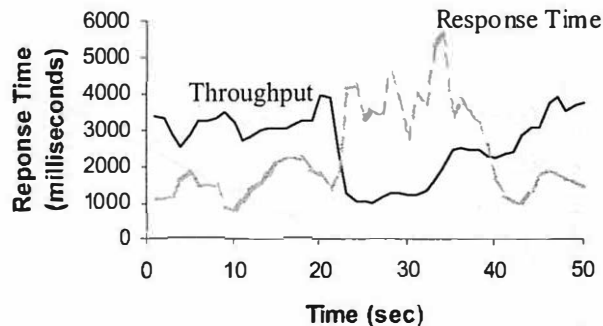


Figure 11: Recovery time for an Unexpected Death  
This graph shows the recovery process for a two-node web server with one failure at time 20. A new clone takes over the traffic in six seconds, and recovery completes in about 20 seconds.

ter, and integrated connection management for high availability and online evolution.

## 5.2 High Availability

Our goal for high availability is to show that users have a high probability of success, and that we can provide independent retry for a given query or connection. It is not our goal to provide fault-tolerant *connections*. Rather, we forfeit active connections on lost nodes, but retries should automatically locate another node and work correctly (by using shared replicated state).

Figure 11 shows the recovery after an unexpected death in a two-node web server. A third clone took over the affected traffic within 6 seconds, and the overall server was fully recovered in about 20 seconds. Response time increased by several seconds during the recovery process, and active connections on the dead node were lost.

In the case of graceful shutdown, Ninja normally does not drop any connections. This case is covered shortly under the discussion of online evolution, which uses controlled shut downs to upgrade a running service without drops.

## 5.3 Graceful Degradation

At the service level, we support several strategies for graceful degradation. The goal is to react gracefully to offered loads that exceed capacity. In practice, peak loads can be 5x the average load, making it impractical to provision for peak load [Mov99]. Even with overprovisioning, 10x load spikes still occur [WS00].

The default strategy is simply to reject new connections when the service is saturated. This preserves the maximum throughput, but is not all that graceful. We provide three strategies that exploit service-level knowledge, via partitions, to degrade more gracefully.

The first strategy is simply to prioritize partitions and assign separate resources for each partition. This enables low-priority partitions to be overloaded without

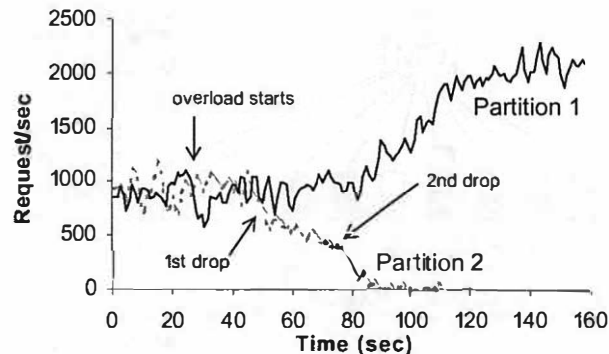


Figure 12: Prioritized Admission Control (Overload)  
We start with a 2-node clone group with two partitions, 1 and 2; Partition 1 has higher priority. Initially both clones handle both partitions. Overload is detected by one of the nodes, which initiates overload mode. At "1st drop" the CM drops half the traffic to the lower priority partition; after a second drop, Partition 2 is not admitted at all and Partition 1 throughput doubles.

affecting high-priority connections, and enables independent throughput asymptotes and overprovisioning ratios. This was shown in Figure 2, in the CM section.

The second strategy, shown in Figure 12, is to prioritize partitions and drop low-priority connections during overload. We refer to this as prioritized admission control. This ensures that under overload the most important connections (or users) are handled first, potentially to the exclusion of lower priority connections. An improvement would be to drop connections probabilistically based on the priority of the partition, but this can essentially be done by first using different resources for each partition, and then dropping connections independently as each partition becomes overloaded.

As discussed in Section 3.3.2, Ninja sheds excess connections by sending them to a "drop" clone. We have not fully explored the power of this mechanism, which could be service specific to enable very fine-grain admission control, since the drop clone could decide on a case-by-case basis to handle some connections.

The third strategy we employ for graceful degradation is to try to serve more requests, but in a degraded form, which is possible because clones *know* that they are in overload mode. For example, a web server might serve generic versions of pages rather than personalized versions. The degraded service moves out the absolute scale limit, at which point further degradation using one of the first two strategies would have to take place.

## 5.4 Online Evolution

Online evolution is enabled by our ability to shut down clones gracefully, without dropped connections. Figure 13 shows online evolution between two versions of a three-node web server. To upgrade a node, the



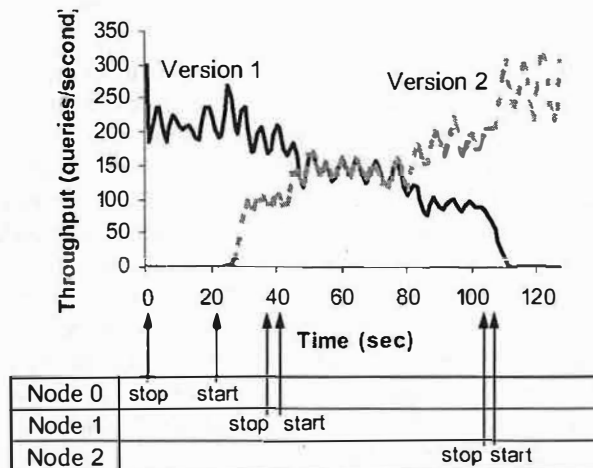


Figure 13: Online Evolution for a 3-node Web Server. Each node starts with Version 1 and is then gracefully shut down and restarted with Version 2. No connections are dropped during the transition. The first restart takes longer due to creation of the clone group for Version 2.

infrastructure first updates the CM to stop sending in new connections to the Version 1 clone. Next, Ninja starts a Version 2 clone on the node, which begins receiving new connections. The Version 1 clone exits once all established connections have been serviced. By repeating this process on all nodes in sequence, we can upgrade the entire cluster with no downtime and no dropped connections.

Note that the two versions typically coexist on the same node for some time while the Version 1 clone finishes servicing existing connections. This is possible because Ninja's virtualization of resources prevents the two versions from interfering with each other (other than performance).

## 5.5 Range of Semantics

Our support for a range of data consistency semantics comes primarily from the CHT and from the ability to build service-specific file systems. The simplest form of this is choosing non-replicated storage, which is possible with both the CHT and the file system toolkit. We have found this useful for local file caches in some of our web server implementations and in the Ninja version of Napster (not discussed).

Figures 3 and 4 show that we can reduce lock contention if we accept updates that have an inconsistent ordering across replicas (using "weak apply"). This approach can achieve five times the throughput with two replicas under heavy contention. Our primary use for this so far has been to maintain unordered lists, which are useful in NinjaMail (since internal message order is not critical), and in various membership lists, such as members of a chat room.

Although not discussed, we have also exploited

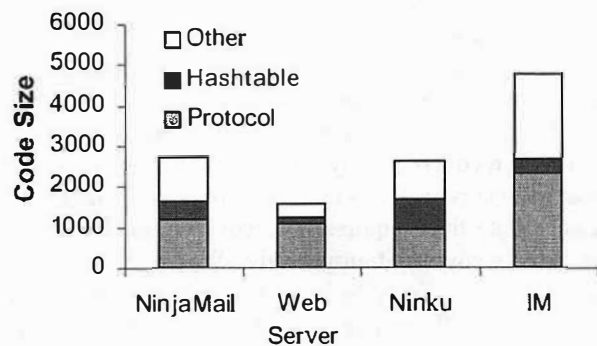


Figure 14: Code Size for Services in Lines (Java)

delayed disk writes in the CFS to improve latency and throughput at the expense of a small window for lost data (much like NFS updates). Similarly, the CHT allows two independent memory copies to be considered "durable", rather than the more strict definition of two copies on disk. In the former case, the disks are updated shortly thereafter in the style of group commit.

## 5.6 Service Authoring

Overall, we found it hard to write the underlying mechanisms, but easy to write the services, which fulfills our primary goal. Figure 14 shows the code size for four applications. The Ninku application, which was not discussed, is the Ninja version of the Napster service. In comparison, the Ninja infrastructure code is about 20,000 lines of code, not counting various third-party libraries used by the CHT and CFS. These services are remarkably small given that they are full-fledged robust services. For example, the Porcupine scalable e-mail server [SBL99] is about 30,000 lines by itself. Both the e-mail and web servers seem to be about one-tenth the size of comparable stand-alone applications.

The primary burden that we did not lighten is the difficulty of authoring protocol code, which presents an obvious place for future work. We also found event-based programming, used for most internal modules and some services to be harder than using threads.

One other important point is that none of these applications require *any* code for high availability, online evolution or graceful degradation, although some may have a few configuration lines to define partitions (if used).

## 6 Discussion

In this section, we review the principles behind Ninja and discuss related and future work.

The programming model has three important principles. First, we want to exploit the natural parallelism of Internet services. The two advantages of this approach are that applications fit naturally and that we can ban more general types of concurrency, which are historically hard to get right and require unknown resources



for correctness. In particular, services do not create or manage threads.

Second, like databases, we want to hide the complexity of fault tolerance, persistence and scalability.

Third, the explicit use of shared state allows us to simplify recovery greatly. The invariant is that anything that must survive faults must be kept in the shared state. Local state thus requires no recovery, and the shared state is recovered transparently. We apply the same principle recursively to build the shared state mechanisms: we differentiate storage clones, which require recovery, from all other clones, which do not. It is the clean recovery story that allows us to provide high availability, online evolution, and dynamic resizing and remapping of clone groups.

We have also pursued a bottom-up approach that provides a range of semantics. For example, the 2PC library and CFS are really tools that are used to simplify building complex systems. The primary difference between the CFS and a traditional file systems is exactly that the CFS is a toolbox with a more appropriate interface for authoring services that need replicated persistent storage. Even the CHT is used as a tool to build the cluster-based file system of NinjaMail. Similarly, we try to make these tools configurable to enable tradeoffs among performance, consistency and replication. The “weak apply” function is the best example of this.

Connection management seems fundamental to robust Internet services. In general, there must at least be a dynamic mapping between external names and currently working internal nodes; otherwise failures are visible to clients. Partitions enable application input into how the CM should prioritize connections. This is essentially a use of static type information (partitions are usually defined statically) to enable run-time optimization during overload. They also provide better cache affinity for all kinds of “front end” clone groups.

The CHT exploits the use of a narrow interface to simplify the maintenance of consistency. Unlike a shared address space, the CHT can only be accessed via method calls and thus only needs to ensure consistency at these points. This is most noticeable in the ability to support atomic updates, as a hash table value is simply not visible during updates.

## 6.1 Related Work

Lightweight recoverable virtual memory [MMK+94] provides an integrated approach to in-memory data structures with durability. It could be used to implement the non-replicated versions of our shared data structures, but does not support replication or 2PC.

The traditional way to simplify persistent applications is to store all data in a DBMS and use a declarative query language for all access and updates. We intention-

ally desire a “navigational” rather than relational interface for better integration with the rest of the service, which is navigational. Databases also focus on consistency under faults at the expense in practice of availability, where we explicitly provide a range of semantics and tradeoffs. We find that availability is often more important for Internet services than strict consistency. DBMSs also provide a large whole solution with little ability to customize semantics or make tradeoffs. In contrast, we may decide to consider something committed if it is in memory on two independent nodes, and only later move objects to disk, which increases throughput for updates. We believe services should use a combination of our techniques and DBMS solutions.

Object-oriented databases, such as Thor [LAC+96] or Persistent Java [ADJ+96], share our use of controlled interfaces, and can implement all of our shared data structures, although they are more heavyweight and generally don’t offer a range of semantics. They are strictly more powerful, with support for transactions and nested objects. We also find power in our “toolbox” approach that has allowed us to build a range of persistent data structures out of logging, 2PC, and the apply function. We also depend on and exploit our partition-free, high-performance network (typical for a cluster).

Application servers, such as BEA’s WebLogic [BEA01], provide persistent shared state by wrapping navigational structures around a relational database. These servers also target large-scale highly available services and were developed concurrently. Application servers typically also provide integration with legacy systems. Ninja provides better support for in-memory data structures, variable semantics, graceful degradation and online evolution. Use of Ninja’s techniques would complement these servers’ use of RDBMS systems, and one vendor is incorporating some of our techniques.

“Layer 7” switches, such as the Foundry switch that we use, provide some aspects of connection management, as does HACC [ZBCS99]. In particular, they can provide load balancing and basic partitioning by URL. The primary advantage of Ninja is the integration of the control of the manager into the framework. We dynamically reconfigure the switches as clone groups change, and we provide integrated support for online evolution and graceful degradation. In fact, our dynamic use of these switches was clearly novel, as we uncovered many new bugs in production hardware that we had to work around.

The Porcupine mail server [SBL99] shares the goals of scalability and availability, and even some of the techniques for replication and scalability. However, Porcupine is a single application rather than a framework. The existence of Ninja makes it easy to write Porcupine: NinjaMail has about one-tenth the code size of Porcu-

pine for similar functionality and robustness. In addition, Ninja is more efficient and allows a wider range of performance tradeoffs than were present in Porcupine.

The TACC framework [FGB97] is a predecessor to this work and shares most of our goals, but does not address persistent shared state, which is the hardest part. It also uses application-specific front ends to do connection management, which we avoid.

The Ninja project [GWv+01] that led to this work has a much broader scope, and includes support for distributed systems built on top of clusters, which we refer to as “bases” in the overall architecture. Some of the additional pieces include support for proxies and end devices, such as laptops, phones, or PDAs; support for paths that connect these elements; security; and OS and proxy support for small devices. There are also papers that cover subsets of the work here in greater detail, including the CHT [GBH+00,Grb00] and SEDA [WCB01].

## 6.2 Future Work

There are at least three key areas of future work: ease of authoring, ease of use, and support for shared state. Section 5.6 covered some enhancements to ease authoring.

To simplify ease of use, there is much we could do to automate online evolution and graceful degradation. Evolution should have an explicit publishing process and a way to revert to the previous version easily. Graceful degradation is mostly automated, but is still very service specific. We don’t help much with how a service should define partitions or trade off quality and performance. We could also use a unified way to test overload conditions and in general administer running services.

Our support for shared state should evolve to include true transactions rather than the atomic actions that we support now. This is quite a bit harder and the current set has proven very useful as is. There is also more we can do with the interaction between lock contention and 2PC, as discussed in Section 3.4.1. Finally, our recovery code remains immature due to the difficulty of thorough testing. However, it is exactly the complexity of recovery code that makes it so valuable to build once for the framework, rather than separately for each service. The automation of recovery is the most valuable aspect of the Ninja framework to service authors.

## 7 Summary

Ninja defines a new programming model and then uses the model to simplify the implementation of complex network services. The model exploits the natural parallelism of large-scale services and hides the complexity of threads, locks, shared state, recovery, load balancing and graceful degradation. It provides several

invariants that greatly simplify service authoring:

- Each service has its own virtual cluster, which may vary in size transparently over time. We have shown linear scalability up to 100 nodes for toy applications and to 30 nodes for the e-mail server.
- Services can have many shared namespaces. Each namespace provides strongly consistent shared state across the nodes of the service.
- Shared state can be persistent and highly available with automatic recovery from faults.
- Concurrency is implicit in the programming model, which avoids the creation and management of threads in applications. Atomicity is provided by the shared state primitives and by isolation of namespaces and local state.
- Connections and external names are managed automatically for load balancing and fault tolerance.
- Overload is detected automatically, which initiates graceful degradation as needed.
- The CM enables online evolution and graceful degradation without help from service authors. They may use partitions for fine-grain control of both quality of service and graceful degradation.
- The CM and highly available shared state together enable highly available services.

Because of these powerful invariants, Ninja services remain remarkably simple despite being scalable, highly available and persistent. We have been able to write several real services using Ninja, including instant messaging, a Napster-like file sharing system, and scalable web and e-mail servers. In all cases, the code for the service was small and relatively simple (e.g. no recovery or logging code). In the case of e-mail, we achieved a ten times reduction in code size for a comparable scalable server by using Ninja.

---

**Acknowledgments:** We would like to thank Amin Vahdat and the referees for their helpful feedback.

- [ADJ+96] M. Atkinson, L. Daynes, M. Jordan, T. Printezis and S. Spence. “An Orthogonally Persistent Java.” *ACM SIGMOD Record*, 25(4), December 1996.
- [Apo01] The Apache Web Server. <http://www.apache.org>.
- [BEA01] The BEA WebLogic Server Datasheet. <http://www.bea.com>
- [DGNP88] F. Darcma, D. A. George, V. A. Norton, and G. F. Pfister. “A single-program-multiple-data computational model for epex/fortran.” *Parallel Computing*, 5(7), 1988.
- [FB99] A. Fox and E. Brewer. “Harvest, Yield, and Scalable Tolerant Systems.” *Proc. of HotOS-VII*. March 1999.
- [FGB97] A. Fox, S. D. Gribble, Y. Chawathe and E. Brewer.

- "Scalable Network Services" *Proc. of the 16th SOSF*, St. Malo, France, October 1997.
- [Fou01] Foundry Networks Scrverlon Switch. <http://www.foundrynet.com/>
- [GBH+00] S. Gribble, E. Brewer, J. M. Hellerstein, and D. Culler. "Scalable, Distributed Data Structures for Internet Service Construction." *Proc. of OSDI 2000*, October 2000.
- [GR97] J. Gray and A. Reuter. *Transaction Processing*. Morgan-Kaufman, 1997
- [Gri00] S. Gribble. *A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction*. Ph.D. Dissertation, UC Berkeley, September 2000.
- [GWv+01] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. "The Ninja Architecture for Robust Internet-Scale Systems and Services." *Journal of Computer Networks*, March 2001.
- [Her91] Maurice Herlihy. *A Methodology for Implementing Highly Concurrent Data Objects*. Technical Report CRL 91/10. Digital Equipment Corporation, October 1991.
- [Hu00] J. Hu. "AOL instant messaging rivals file complaint with FCC." CNET News.com. <http://news.cnet.com/news/0-1005-200-1755834.html>, April 25, 2000.
- [HT93] D. Hillis and L. W. Tucker. "The CM-5 Connection Machine: a scalable supercomputer." *Communications of the ACM*, 36(11), pp. 31-40, November 1993.
- [HW87] M. P. Herlihy and J. M. Wing. "Axioms for concurrent objects." *Proc. of the 14th SOSF*, pp. 13-26, January 1987.
- [LAC+96] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day and L. Shrira. "Safe and efficient sharing of persistent objects in Thor." *Proc. of ACM SIGMOD*, pp. 318-329, 1996.
- [Lar00] J. Larus. *Enhancing server performance with Staged-Server*. <http://www.research.microsoft.com/~larus/Talks/StagedServer.ppt>, October 2000.
- [MMK+94] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, J. J. Kistler. "Lightweight Recoverable Virtual Memory." *ACM Transactions on Computer Systems*, 12(1), pp. 33-57, February 1994.
- [Mov99] MovieFone Corporation. "MovieFone Announces Preliminary Results from First Day of Star Wars Advance Ticket Sales." Press Release, May 12, 1999.
- [PAB+98] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. "Locality-Aware Request Distribution in Cluster-based Network Servers." *Proc. of ASPLOS 1998*. October 1998.
- [PDZ99] V. S. Pai, P. Druschel, and W. Zwaenepoel. "Flash: An efficient and portable Web server." *Proc. of the 1999 Annual USENIX Technical Conference*, June 1999.
- [RV89] E. Roberts and M. Vandevoorde. *Work crews: An abstraction for controlling parallelism*. DEC SRC Technical Report 42, Palo Alto, California, 1989.
- [SBL99] Y. Saito, B. Bershad and H. Levy. "Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-based Mail Service." *Proc. of the 17<sup>th</sup> SOSF*. October 1999.
- [Sco96] S. L. Scott. "Synchronization and Communication in the T3E Multiprocessor." *Proc. of ASPLOS 1996*, October 1996.
- [SPEC99] Standard Performance Evaluation Corporation. *The SPECweb99 Benchmark*. <http://www.spec.org/osg/web99>.
- [WM01] M. Williams and M. Berger. "Two days late, Hotmail gets an upgrade." *InfoWorld*, July 19, 2001.
- [WCB01] M. Welsh, D. Culler and E. Brewer. "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services." *Proc. of the 18th SOSF*. October, 2001.
- [WS00] L. A. Wald and S. Schwarz. "The 1999 Southern California Seismic Network Bulletin." *Seismological Research Letters*, 71(4), July/August 2000.
- [Yah01] Yahoo! Inc. "Yahoo! Introduces Yahoo! Mail - Business Edition." Press Release, October 15, 2001.
- [ZBCS99] X. Zhang, M. Barricatos, J. B. Chen, and M. Seltzer. "HACC: An Architecture for Cluster-Based Web Servers." *Proc. of the 3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999.

# Using Cohort Scheduling to Enhance Server Performance

James R. Larus and Michael Parkes

{larus, mparkes}@microsoft.com

Microsoft Research

One Microsoft Way

Redmond, WA 98052

## Abstract

A server application is commonly organized as a collection of concurrent threads, each of which executes the code necessary to process a request. This software architecture, which causes frequent control transfers between unrelated pieces of code, decreases instruction and data locality, and consequently reduces the effectiveness of hardware mechanisms such as caches, TLBs, and branch predictors. Numerous measurements demonstrate this effect in server applications, which often utilize only a fraction of a modern processor's computational throughput.

This paper addresses this problem through *cohort scheduling*, a new policy that increases code and data locality by batching the execution of similar operations arising in different server requests. Effective implementation of the policy relies on a new programming abstraction, *staged computation*, which replaces threads. The *StagedServer* library provides an efficient implementation of cohort scheduling and staged computation. Measurements of two server applications written with this library show that cohort scheduling can improve server throughput by as much as 20%, by reducing the processor cycles per instruction by 30% and L2 cache misses by 50%.

## 1 Introduction

A server application is a program that manages access to a shared resource, such as a database, mail store, file system, or web site. A server receives a stream of requests, processes each, and produces a stream of results. Good server performance is important, as it determines the latency to access the resource and constrains the server's ability to handle multiple requests. Commercial servers, such as databases, have been the focus of considerable research to improve the underlying hardware, algorithms, and parallelism, as well as considerable development to improve their code.

Much of the hardware effort has concentrated on the memory hierarchy, where rapidly increasing processor speed and parallelism and slowly declining memory access time created a growing gap that is a major performance bottleneck in many systems. In recent proces-

sors, loading a word from memory can cost hundreds of cycles, during which three to four times as many instructions could execute. High performance processors attempt to alleviate this performance mismatch through numerous mechanisms, such as caches, TLBs, and branch predictors [27]. These mechanisms exploit a well-known program property—spatial and temporal reuse of code and data—to keep at hand data that is likely to be reused quickly and to predict future program behavior.

Server software often exhibits less program locality and, consequently achieves poorer performance, than other software. For example, many studies have found that commercial database systems running on-line transaction processing (OLTP) benchmarks incur high rates of cache misses and instruction stalls, which reduce processor performance to as low as a tenth of its peak potential [4, 9, 20]. Part of this problem may be attributable to database systems' code size [28], but their execution model is also responsible.

These systems are structured so that a process or thread runs for a short period before invoking a blocking operation and relinquishing control, so processors execute a succession of diverse, non-looping code segments that exhibit little locality. For example, Barroso et al. compared TPC-B, an OLTP benchmark whose threads execute an average of 25K instructions before blocking, against TPC-D, a compute-intensive decision-support system (DSS) benchmark whose threads execute an average of 1.7M instructions before blocking [9]. On an AlphaServer 4100, TPC-B had an L2 miss rate of 13.9%, an L3 miss rate of 2.7%, and overall performance of 7.0 cycles per instruction (CPI). By contrast, TPC-D had an L2 miss rate of 1.2%, an L3 miss rate of 0.32%, and a CPI of 1.62.

Instead of focusing on hardware, this paper takes an alternative—and complementary—approach of modifying a program's behavior to improve its performance. The paper presents a new, user-level software architecture that enhances instruction and data locality and increases server software performance. The architecture consists of a scheduling policy and a programming model. The policy, *cohort scheduling*, con-

secutively executes a cohort of similar computations that arise in distinct requests on a server. Computations in a cohort, because they are at roughly the same stage of processing, tend to reference similar code and data, and so consecutively executing them improves program locality and increases hardware performance. *Staged computation*, the programming model, provides a programming abstraction by which a programmer can identify and group related computations and make explicit the dependences that constrain scheduling. Staged computation, moreover, has the additional benefits of reducing concurrency overhead and the need for expensive, error-prone synchronization.

We implemented this scheduling policy and programming model in a reusable library (*StagedServer*). In two experiments, one with an I/O-intensive server and another with a compute-bound server, code using *StagedServer* performed significantly better than threaded versions. *StagedServer* lowered response time by as much as 20%, reduced cycles per instruction by 30%, and reduced L2 cache misses by more than 50%.

The paper is organized as follows. Section 2 introduces cohort scheduling and explains how it can improve program performance. Section 3 describes staged computation. Section 4 briefly describes the *Staged-Server* library. Section 5 contains performance measurements. Section 6 discusses related work.

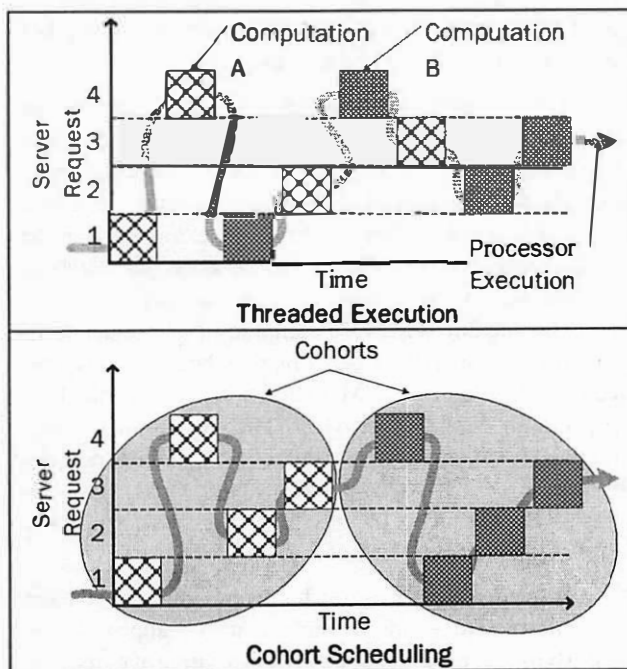


Figure 1. Cohort scheduling in operation. Shaded boxes indicate different computations performed while processing requests on a server. Cohort scheduling reorders the computations, so that similar ones execute consecutively on a processor, which increases program locality and processor performance.

## 2 Cohort Scheduling

Cohort scheduling is a technique for organizing the computation in server applications to improve program locality. The key insight is that distinct requests on a server execute similar computations. A server can defer processing a request until a *cohort of computations* arrive at a similar point in their processing and then execute the cohort consecutively on a processor (Figure 1).

This scheduling policy increases opportunities for code and data reuse, by reducing the interleaving of unrelated computations that causes cache conflicts and evicts live cache lines. The approach is similar to loop tiling or blocking [19], which restructures a matrix computation into submatrix computations that repeatedly reference data before turning to the next submatrix. Cohort scheduling, however, is a dynamic process that reorganizes a series of computations on items in an input stream, so that similar computations on different items execute consecutively. The technique applies to uniprocessors and multiprocessors, as both depend on program locality to achieve good performance.

Figure 2 illustrates the results of a simple experiment that demonstrates the benefit of cohort scheduling on a uniprocessor. It reports the cost, per call, of executing different sized cohorts of asynchronous writes to random blocks in a file. Each cohort ran consecutively on a system whose cache and branch table buffer had been flushed. As the cohort increased in size, the cost of each call decreased rapidly. A single call consumed 109,000 cycles, but the average cost dropped 68% for a cohort of 8 calls and 82% for a cohort of 64 calls. A direct measure of locality, L2 cache misses, also improved dramatically. With a cohort of 8 calls, L2 misses per call dropped to 17% of the initial value and further declined to 4% with a cohort of 64 calls. These improvements required no changes to the operating system code; only reordering operations in an application. Further improvement requires reductions in OS self-conflict misses (roughly 35 per system call), rather than amortizing the roughly 1500 cold start misses.

### 2.1 Assembling Cohorts

Cohort scheduling is not irreparably tied to staged computation, but many benefits may be lost if a programmer cannot explicitly form cohorts. For example, consider transparently integrating cohort scheduling with threads. The basic idea is simple. A modified thread scheduler identifies and groups threads with identical next program counter (nPC) values. Threads starting at the same point are likely to execute similar operations, even if their behavior eventually diverges. The scheduler runs a cohort of threads with identical nPCs before turning to the next cohort.



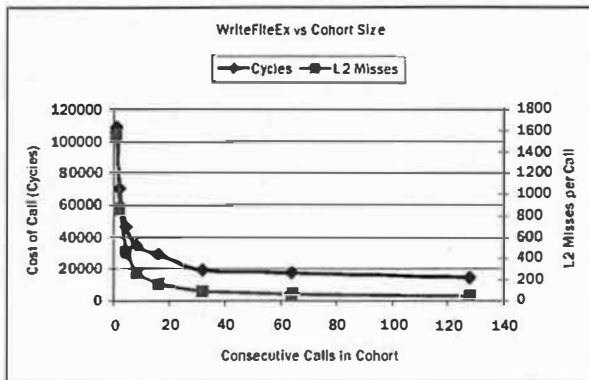


Figure 2. Performance of cohorts of WriteFileEx system calls in Window 2000 Advanced Server (Dell Precision 610 with an Intel Pentium III processor). The chart reports the cost per call—in processor cycles and L2 cache misses—of an asynchronous write to a random 4K block in a file.

It is easy to believe that this scheme could sometimes improve performance, and it requires only minor changes to a scheduler and no changes to applications. It, however, has clear shortcomings. In particular, nPC values are a coarse and indirect indicator of program behavior. Only threads with identical nPCs end up in a cohort, which misses many pieces of code with similar behavior. For example, several routines that access a data structure might belong in a cohort. Simple extensions to this scheme, such as using the distance between PCs as a measure of similarity, have little connection to logical behavior and are perturbed by compiler linking and code scheduling. Another disadvantage is that cohorts start after blocking system calls, rather than at application-appropriate points. In particular, compute-intensive applications or programs that use asynchronous I/O cannot use this scheme, as they do not block.

To correct these shortcomings and properly assemble a cohort, a programmer must delimit computations and identify the ones that belong in a cohort. Staged computation provides a programming abstraction that neatly captures both dimensions of cohorts.

### 3 Staged Computation

Staged computation is a programming abstraction intended to replace threads as the construct underlying concurrent or parallel programs. Stages offer compelling performance and correctness advantages and are particularly amenable to cohort scheduling. In this model, a program is constructed from a collection of *stages*, each of which consists of a group of exported operations and private data. An operation is an asynchronous procedure call, so its invocation, execution, and reply are decoupled. Moreover, a stage has *scheduling autonomy*, which enables it to control the order and concurrency with which its operations execute.

A stage is conceptually similar to a class in an object-based language, to the extent that it is a program structuring abstraction providing local state and operations. Stages, however, differ from objects in three major respects. First, operations in a stage are invoked asynchronously, so that a caller does not wait for a computation to complete, but instead continues and rendezvous later, if necessary, to retrieve a result. Second, a stage has autonomy to control the execution of its operations. This autonomy extends to deciding when and how to execute the computations associated with invoked operations. Finally, stages are a control abstraction used to organize and process work, while objects are a data representation acted on by other entities, such as functions, threads, or stages.

A stage facilitates cohort scheduling because it provides a natural abstraction for grouping operations with similar behavior and locality and the control autonomy to implement cohort scheduling. Operations in a stage typically access local data, so that effective cohort scheduling only requires a simple scheduler that accumulates pending operations to form a cohort.

Stages provide additional programming advantages as well. Because they control their internal concurrency, they promote a programming style that reduces the need for expensive, error-prone explicit synchronization. Stages, moreover, provide the basis for specifying and verifying properties of asynchronous programs. This section briefly describes the staged programming model. Section 4 elaborates an implementation in a C++ class library.

#### 3.1 Stage Design

Programmers group operations into a stage for a variety of reasons. The first is to regulate access to program state (“static” data) by wrapping it in an abstract data type. Operations grouped this way form an obvious cohort, as they typically have considerable instruction and data locality. Moreover, a programmer can control concurrency in a stage to reduce or eliminate synchronization for this data (Section 3.4).

The second reason is to group logically related operations to provide a well-rounded and complete programming abstraction. This reason may seem less compelling than the first, but logically related operations frequently share code and data, so collecting them in a stage identifies operations that could benefit from cohort scheduling.

The third is to encapsulate program control logic in the form of a finite-state automaton. As discussed below, a stage’s asynchronous operations easily implement the reactive transitions in an event-driven state machine.



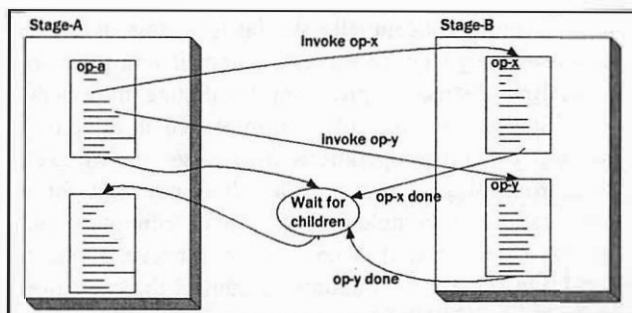


Figure 3. Example of stages and operations. Stage-A runs *op-a*, which invokes two operations in Stage-B and waiting until they complete before running *op-a*'s continuation.

In practice, designing a program with stages focuses on partitioning the tasks into sub-tasks that are self-contained, have considerable code and data locality, and have logical unity. In many ways, this process is the control analogue of object-oriented design.

## 3.2 Operations

Operations are asynchronous computations exported by a stage. Invocation of an operation only requires its eventual execution, so the invoker and operation run independently. When an operation executes, it can invoke any number of child operations on any stage, including its own. A parent can wait for its children to finish, retrieve results from their computation, and continue processing. Figure 3 shows an operation (*op-a*) running in *Stage-A* that invokes two operations (*op-x* and *op-y*) in *Stage-B*, performs further computation, and then waits for its children. After they complete and return their results, *op-a* continues execution and processes the children's results.

The code within an operation executes sequentially and can invoke both conventional (synchronous) calls and asynchronous operations. However, once started, an operation is non-preemptible and runs until it relinquishes the processor. Programmers, unfortunately, must be careful not to invoke blocking operations that suspend the thread running operations on a processor. An operation that relinquishes the processor to wait for an event—such as asynchronous I/O, synchronization, or operation completion—instead provides a *continuation* to be invoked when the event occurs [14].

A continuation consisting of a function and enough saved state to permit the computation to resume at the point at which it suspended. *Explicit continuations* are the simplest and least costly approach, as an operation saves only its live state in a structure called a *closure*. The other alternative, *implicit continuations*, requires the system to save the executing operation's stack, so that it can be resumed. This scheme, similar to fibers, simplifies programming, at some performance cost [2].

Asynchronous operations provide low-cost parallelism, which enables a programmer to express and exploit the concurrency within an application. The overhead, in time and space, of invoking an operation is close to a procedure call, as it only entails allocating and initializing a closure and passing it to a stage. When an operation runs to completion, it does not require its own stack or an area to preserve processor state, which eliminates much of the cost of threads. Similarly, returning a value and re-enabling a continuation are simple, inexpensive operations.

## 3.3 Programming Styles

Staged computation supports a variety of programming styles, including software pipelining, event-driven state machines, bi-directional pipelines, and fork-join parallelism. Conceptually, at least, stages in a server are arranged as a pipeline in which requests arrive at one end and responses flow from the other. This form of computation is easily supported by representing a request as an object passed between stages. Linear pipelining of this sort is simple and efficient, because a stage retains no information on completed computations.

However, stages are not constrained to this linear style. Another common programming idiom is bi-directional pipelining, which is the asynchronous analogue of call and return. In this approach, a stage passes subtasks to one or more other stages. The parent stage eventually suspends its work on the request, turns its attention to other requests, and resumes the original computation when the subtasks produce results. This style requires that an operation be broken into a series of subcomputations, which run when results appear. With explicit continuations, a programmer partitions the computation by hand, although a compiler could easily produce this code, which is close to the well-known continuation-passing style [6, 12]. With implicit continuations, a programmer only needs to indicate where the original computation suspends and waits for the subtasks to complete.

A generalization of this style is event-driven programming, which uses a finite state automaton (FSA) to control a reactive system [26, 29]. The FSA logic is encapsulated in a stage and is driven by external events, such as network messages and I/O completions, and internal events from other asynchronous stages. An operation's closure contains the FSA state for a particular server request. The FSA changes state when a child operation completes or external events arrive. These transitions invoke computations associated with edges in the FSA. Each computation runs until it blocks and specifies the next state in the FSA.

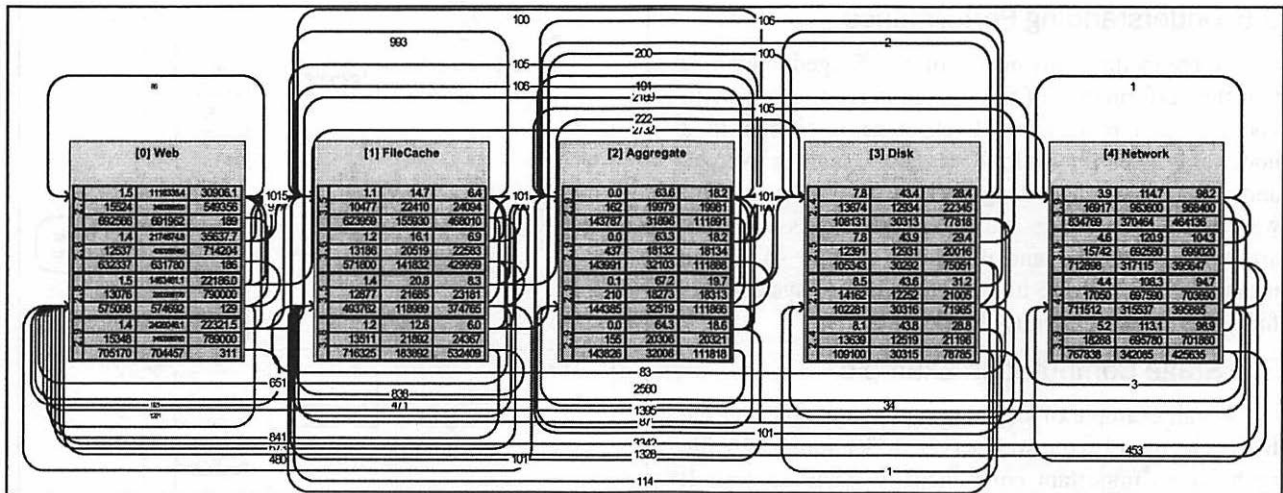


Figure 4 Profile of staged web server (Section 5.1). The performance metrics for each stage are broken down by processor (the system is running on four processors). The first column is the average queue length. The second column contains three metrics on operations at the stage: the quantity, the average wait time (millisecond), and the maximum wait time. The third column contains corresponding metrics for operations that are suspended and restarted. The fourth column contains corresponding metrics for completed operations. The numbers on arcs are the number of operations started or restarted between two stage-processor pairs.

For example, the web server used in Section 5.1 is driven by a control-logic stage consisting of a FSA with fifteen states. The FSA describes the process by which a HTTP GET request arrives and is parsed, the referenced file is found in the cache or on disk, the file blocks are read and transmitted, and the file and connection are closed.

Describing the control logic of a server as a FSA opens the possibility of verifying many properties of the entire system, such as deadlock freedom, by applying techniques, such as model checking [15, 22], developed to model and verify systems of communicating FSAs.

### 3.4 Scheduling Policy Refinements

The third attribute of a stage is scheduling autonomy. When a stage is activated on a processor, the stage determines which operations execute and their order. This scheduling freedom allows several refinements of cohort scheduling to reduce the need for synchronization. In particular, we found three policies useful:

- An *exclusive stage* executes at most one of its operations at a time. Since operations run sequentially and completely, access to stage-local data does not need synchronization. This type of a stage is similar to a monitor, except that its interface is asynchronous: clients delegate computation to the stage, rather than block to obtain access to a resource. When this strict serialization does not cause a performance bottleneck, this policy offers fast, error-free access to data and a simple programming model. This approach works well for fine-grained operations, as the cost of acquiring and releasing the

stage's mutex can be amortized over a cohort of operations [25].

- A *partitioned stage* divides invocations (based on a key passed as a parameter), to avoid sharing data among operations running on different processors. For example, consider a file cache stage that partitions requests using a hash function on the file number. Each processor maintains its own hash table of in-memory disk blocks. Each hash table is accessed by only one processor, which enhances locality and eliminates synchronization. This policy, which is reminiscent of shared-nothing databases, permits parallel data structures without fine-grain synchronization.
- A *shared stage* runs its operations concurrently on many processors. Since several operations in a stage can execute concurrently, shared data accesses must be synchronized.

Other policies are possible and could be easily implemented within a stage.

It is important keep in mind that these policies are implemented within the more general framework of cohort scheduling. When a stage is activated on a processor, it executes its outstanding operations, one after another. Nothing in the staged model requires cohort scheduling. Rather the programming model and scheduling policy naturally fit together. A stage groups logically related operations that share data and provides the freedom to reorder computations. Cohort scheduling exploits scheduling freedom by consecutively running similar operations.

### 3.5 Understanding Performance

A compelling advantage of the Staged model is that the performance of the system is relatively easy to visualize and understand. Each stage is similar to a node in a queuing system. Parameters, such as average and maximum queue length, average and maximum wait time, and average and maximum processing time, are easily measured and displayed (Figure 4). These measurements provide a good overview of system performance and help identify bottlenecks.

### 3.6 Stage Computation Example

As an example of staged computation, consider the file cache used by the web server in Section 5.1. A file cache is an important component in many servers. It stores recently accessed disk blocks in memory and maps a file identifier and offset to a disk block.

The staged file cache consists of three partitioned stages (Figure 5). The cache is logically partitioned across the processors, so each one manages a unique subset of the files, as determined by the hashed file identifier. Alternatively, for large files, the file identifier and offset can be hashed together, so a file's disk blocks are striped across the table. Within the stage, each processor maintains a hash table that maps file identifiers to memory-resident disk blocks. Since a processor references only its table, accesses require no synchronization and data does not migrate between processor caches.

If a disk block is not cached in memory, the cache invokes an operation on the *I/O Aggregator* stage, whose role is to merge requests for adjacent disk blocks to improve system efficiency. This stage utilizes cohort scheduling in a different way, by accumulating I/O requests in a cohort and combining them into a larger I/O request on the operating system.

The *Disk I/O* stage reads and writes disk blocks. It issues asynchronous system calls to perform these operations and, for each, invokes an operation in the *Event Server* stage describing a pending I/O. This operation suspends until the I/O completes. This stage interfaces the operating system's asynchronous notification mechanism to the staged programming model. It utilizes a separate thread, which waits on an I/O Completion Port that the system uses to signal completion of asynchronous I/O. At each notification, this stage matches an event with a waiting closure, which it re-enables and passes the information from the Completion Port. The *Disk I/O* stage, in turn, returns disk blocks to the I/O Aggregator, which passes them to the *FileCache* stage, where the data are recorded and passed back to the client.

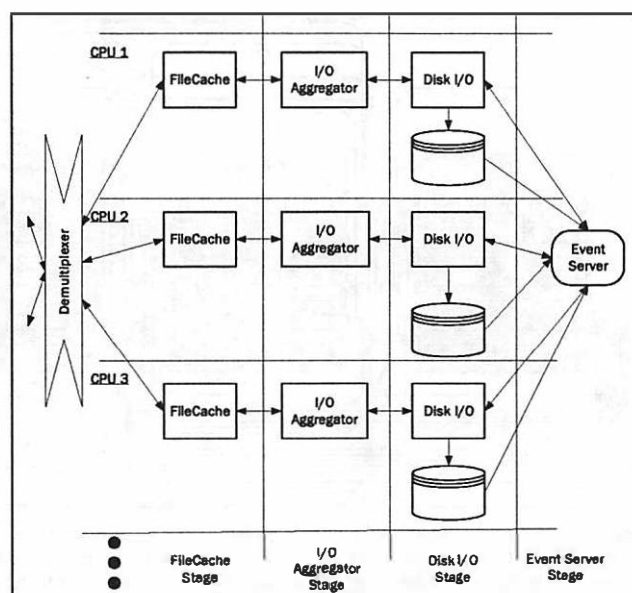


Figure 5. Architecture of staged file cache. Requests for disk blocks are partitioned across processors to avoid sharing the hash table. If a block is not found, it is requested from an I/O aggregator, which combines requests for adjacent blocks and passes them to a disk I/O stage that asynchronously reads the files. When an I/O completes, an event server thread is notified, which passes the completion back to the disk I/O stage.

## 4 StagedServer Library

The StagedServer library is a collection of C++ classes that implement staged computation and cohort scheduling on either a uniprocessor or multiprocessor. This library enables a programmer to define stages, operations, and policies by writing only application-specific code. Moreover, StagedServer implements an aggressive and efficient version of cohort scheduling. This section briefly describes the library and its primary interfaces.

The library's functionality is partitioned between two principal classes. The first is the *Stage* class, which provides stage-local storage and mechanisms for collecting and scheduling operations. The second is the *Closure* class, which encapsulates an operation and its continuations, provides per-invocation state, and supports invoking an operation and returning its result. The fundamental action in a StagedServer system is to invoke an operation by creating and initializing a closure and handing it to a stage.

### 4.1 Stage Class

The *Stage* class is a templated base class that an application uses to derive classes for its various stages. The base class provides the basic functionality for managing closures and for scheduling and executing operations on processors.

### 4.1.1 Scheduling Policy

StagedServer implements a cohort scheduling policy, with enhancements to increase the processor affinity of data. The assignment of operations to processors occurs when an operation is submitted to a stage. By default, an operation invoked by code running on processor  $p$  executes on processor  $p$  in subsequent stages. This affinity policy enhances temporal locality and reduces cache traffic, as the operation's data tend to remain in the processor's cache. However, a program can override the default and execute an operation on a different processor when: the processor to execute the operation is explicitly specified, a stage partitions its operations among processors, or a stage uses load balancing to redistribute operations.

A stage maintains a stack and queue for each processor in the system. In general, operations originating on the local processor are pushed on the stack and operations from other processors are enqueued on the queue. When a stage starts processing a cohort, it first empties its stack in LIFO order, before turning to the queue. This scheme has two rationales. Processing the most recently invoked operations first increases the likelihood that an operation's data will reside in the cache. In addition, the stack does not require synchronization, since it is only accessed by one processor, which reduces the common-case cost of invoking an operation.

### 4.1.2 Processor Scheduling

StagedServer currently uses a simple, wavefront algorithm to supply processors to stages. A programmer specifies an ordering of the stages in an application. In wavefront scheduling, processors independently alternate forward and backward traversals of this list of stages. At each stage, a processor executes operations pending in its stack and queue. When the operations are finished, the processor proceeds to the next stage. If the processor repeatedly finds no work, it sleeps for exponentially increasing periods of time interval. If a processor cannot gain access to an Exclusive stage, because another processor is already working in the stage, the processor skips the stage.

The alternating traversal order in wavefront scheduling corresponds to a common communications pattern, in which a stage passes requests to its successors, which perform a computation and produce a result. It is easy to imagine other scheduling policies, but we have not evaluated them, as this approach works well for the applications we have studied. This topic is worth further investigation.

### 4.1.3 Thresholds

An orthogonal attribute of a stage is a pair of thresholds that force StagedServer to activate a stage if more than a given number of operations are waiting or after a fixed interval. When either situation arises, StagedServer stops the currently running stage (after it completes its operation), runs the threshold-exceeding stage, and then returns to the suspended stage. For simplicity, an interrupting stage cannot be interrupted, so that other stages that exceed their thresholds are deferred until processing returns to original stage. Thresholds are particularly useful for latency-sensitive stages, such as those interacting with the operating system, which must be regularly supplied with I/O requests to ensure that devices do not go idle.

Another useful refinement is a feedback mechanism, by which a stage informs other stages that it has sufficient tasks. These other stages can suspend processing, effectively turning the processor over to the first stage. So far, voluntary cooperation, rather than hard queue limits, has sufficed.

### 4.1.4 Partitioned Data

A partitioned stage typically divides its data, so that the operations running on a processor access only a non-shared portion. Avoiding sharing eliminates the need to synchronize access to the data and reduces the cache traffic that results when data are accessed from more than one processor. The current system partitions a variable—using the well-known technique of privatization [30]—by storing its values in a vector with an entry for each processor. Code uses the processor id to index this vector and obtain a private value.

## 4.2 Closure Class

Closure is a templated base class for defining closures, which are a combination of code and data. StagedServer uses closures to implement operations and their continuations. When an operation is first invoked on a stage, the invoker creates a closure and initializes it with parameter values. Later, the stage executes the operation by invoking one of the closure's methods, as specified by the operation invocation. This method is an ordinary C++ method. When it returns, the method must state whether the operation is complete (and optionally returns a value), if it is waiting for a child to finish, or if it is waiting for another operation to resume its execution.

An operation can invoke operations on other stages—its children. The original operation waits for its children by providing a continuation routine that the system runs when the children finish. This continuation routine is simply another method in the original closure.

The closure passes arguments between a parent and its continuation and results between a child and its parent. This process may repeat multiple times, with each continuation taking on the role of a parent. In other words, these closures are actually multiple-entry closures, with an entry for the original operation invocation and entries for subsequent continuations. In practice, a stage treats these methods identically and does not distinguish between an operation and its continuation.

## 5 Experimental Evaluation

To evaluate the benefits of cohort scheduling and the StagedServer library, we built two prototypical server applications. The first—a web server—is I/O-bound, as its task consists of responding to HTTP GET requests by retrieving files from a disk or file cache and sending them over a network. The second—a publish-subscribe server—is compute bound, as the amount of data transferred is relatively small, but the computation to match an event against a database of subscriptions is expensive and memory-intensive.

### 5.1 I/O-Intensive Server

To compare threads against stages, we implemented two web servers. The first is structured using a thread pool (THWS) and the second uses StagedServer (SSWS). We took care to make the two servers efficient and comparable and to share common code. In particular, both servers use Microsoft Windows's asynchronous I/O operations. The threaded server was organized in a conventional manner as a thread accepting connections and passing them to a pool of 256 worker threads, each of which performs the server's full functionality: parsing a request, reading a file, and transmitting its contents. This server used the kernel's file cache. The SSWS server also can process up to 256 simultaneously requests. It was organized as a control logic stage, a network I/O stage, and the disk I/O and caching stages described in Section 3.6. The parameters were chosen by experimentation and yielded robust performance for the benchmark and hardware configuration.

As a baseline for comparison, we also ran the experiments on Microsoft's IIS web server, which is a highly tuned commercial product. IIS performed better than the other servers, but the difference was small, which partially validates their implementations.

Our test configuration consisted of a server and three clients. The server was Compaq Proliant DL580R containing four 700MHz Pentium III-Xeon processors (2MB L2 cache) and 4GB of RAM. It had eight

10000RPM SCSI3 disks, connected to a Compaq Smart Array controller. The clients ran on Dell PowerEdge 6350s, each containing four 400MHz Pentium II Xeon processors with 1GB of RAM. The clients and server were connected by a dedicated Gigabit Ethernet network and both ran Windows 2000 Server (SPI).

We used the SURGE benchmark, which retrieves web pages, whose size, distributions, and reference pattern are modeled on actual systems [8]. SURGE measures the ability of a web server to process HTTP GET requests, retrieve pages from a disk, and send them back to a client. This benchmark does not attempt to capture the full behavior of a web server, which must handle other types of HTTP requests, execute dynamic content, perform server management, and log data. To increase the load, we run a large configuration, with a web site of 1,000,000 pages (20.1 GB) and a reference stream containing 6,638,449 requests. A SURGE workload is characterized by User-Equivalents (UEs), each of which models one user accessing the web site. We found that we could run up to 2000 UEs per client. All tests were run with the UE workload balanced across the client machines. The reported numbers are for 15 minutes of client execution, starting with a freshly initialized server.

Figure 6 shows the bandwidth and latency of the thread (THWS) and StagedServer (SSWS) servers, and compares them against a commercial web server (IIS). The first chart contains the number of pages retrieved by the clients per second (since requests follow a fixed sequence, the number of pages is a measure of bandwidth) and the second chart contains the average latency, perceived by a client, to retrieve a page.

Several trends are notable. Under light load, SSWS's performance is approximately 6% lower than THWS, but as the load increases, SSWS responds to as many as 13% more requests per unit time. The second chart, in part, explains this difference. SSWS's latency is higher than THWS's latency under light load (by a factor of almost 20), but as the load increases, SSWS's latency grows only 2.3 times, but THWS's latency increases 45 times, to a level equal to SSWS's.

The commercial server, Microsoft's IIS, outperformed SSWS by 4–9% and THWS by 0–22%. Its latency under heavy load was up to 45% better than the other servers' latency.



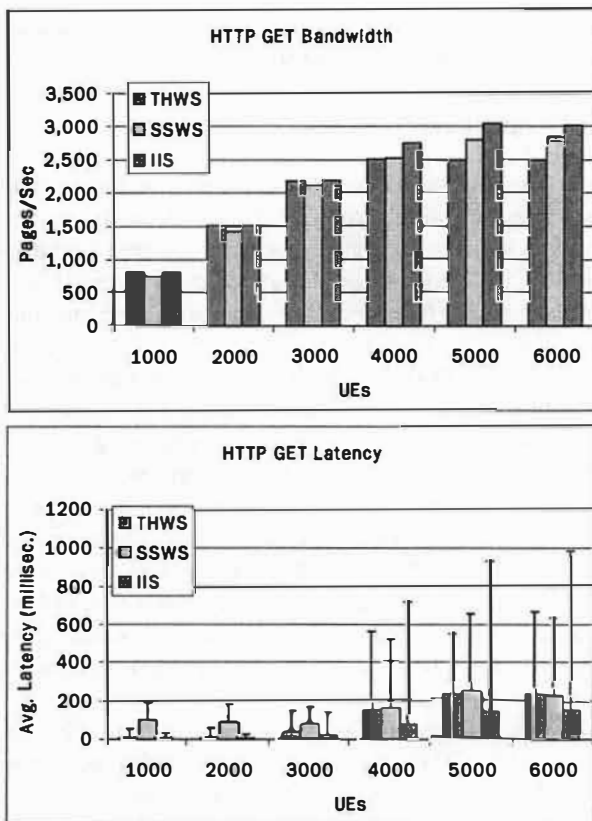


Figure 6. Performance of web servers. These charts show the performance of the threaded server (THWS), StagedServer server (SSWS), and Microsoft's IIS server (IIS). The first records the number of web pages received by the clients per second. The second records the average latency, as perceived by the client, to retrieve a page. The error bars are the standard deviation of the latency.

SSWS performance, which is more stable and predictable under heavy load than the threaded server, is appropriate for servers, in which performance challenges arise as offered load increases. SSWS server's overall performance was relatively better and its processor performance degraded less under load than the THWS server. The improved processor performance was reflected in a measurably improved throughput under load.

## 5.2 Compute-Bound Server

To evaluate the performance of StagedServer on a compute-bound application, we also built a simple publish-subscribe server. The server used an efficient, cache-friendly algorithm to match events against an in-core database of subscriptions [16]. A subscription is a conjunction of terms comparing variables against integer. An event is a set of assignments of values to variables. An event matches a subscription if all of its terms are satisfied by the value assignments in the event.

Both the threaded (THPS) and StagedServer (SSPS) version of this application shared a common

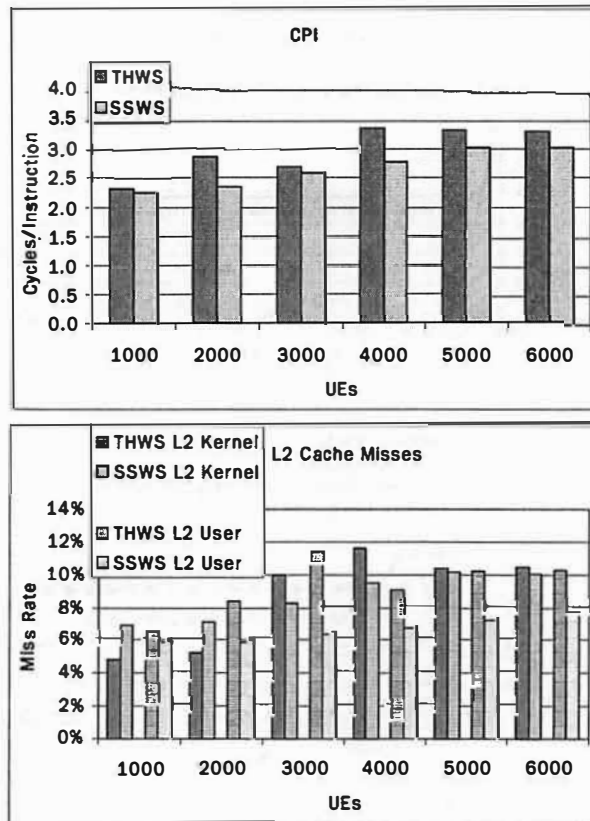


Figure 7. Processor performance of servers. These charts show the processor performance of the threaded (THWS) and StagedServer (SSWS) web server. The first chart shows the cycles per instruction (CPI) and the second shows the rate of L2 cache misses.

publish-subscribe implementation; the only difference between them was the use of threads or stages to structure the computation. The benchmark was the Fabret workload: 1,000,000 subscriptions and 100,000 events. The platform was the same as above.

The response time of the StagedServer version to events was better under load (Figure 8). With four or more clients publishing events, the THPS responded in an average of 0.57 ms to each request. With four clients, SSPS responded in an average time of 0.53 ms, and its response improved to 0.47 ms with 16 or more clients (21% improvement over the threaded version).

In large measure, this improvement is due to improved processor usage (Figure 8). With 16 clients, SSPS averaged 2.0 cycles per instruction (CPI) over the entire benchmark, while THPS averaged 2.7 CPI (26% reduction). Over the compute-intensive event matching portion, SSPS averaged 1.7 CPI, while THPS averaged 2.5 CPI (33% reduction). In large measure, this improvement is attributable to a greater than 50% reduction in L2 caches misses, from 58% of user-space L2 cache requests (THPS) to a 26% of references (SSPS).



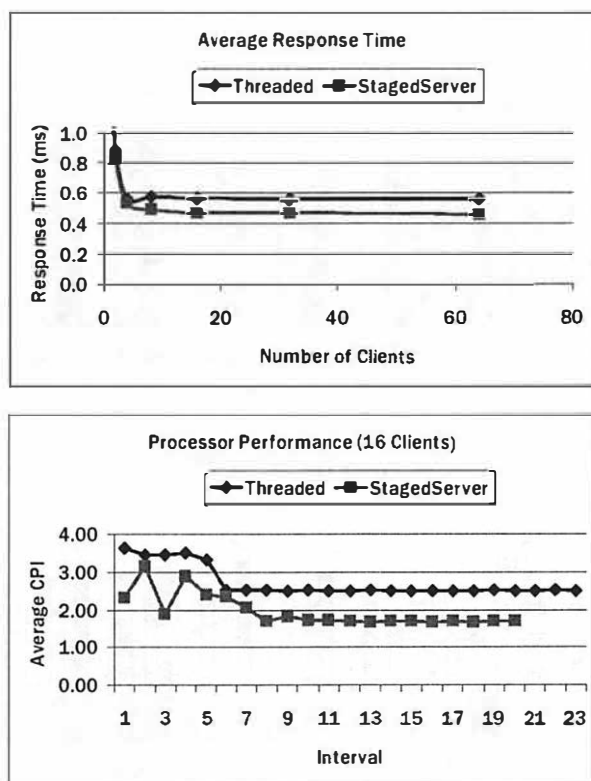


Figure 8. Performance of Publish-Subscribe server. The top chart records the average response time to match publish events against subscriptions. The bottom chart compares average cycles per instruction (CPI) of the thread and StagedServer versions over 25 second intervals. The initial part of each curve is the construction of internal data structures, while the flat part of the curves is the event matching.

This application references a large data structure (approximately 66.7MB for the benchmark). When matching an event against subscriptions, Fabret's algorithm, although cache-efficient, may access a large amount of data, and the particular locations are data dependent. StagedServer's performance advantage results from two factors. First, its code is organized so that only one processor references a given subset of the subscriptions, which reduces the number of distinct locations a processor references, and hence increases the possibility of data reuse. Without this locality optimization, SSPS runs at the same speed as THPS. Second, StagedServer batches cohorts of event matches in this data structure. We measured the benefit of cohort scheduling by limiting cohort size. Cohorts of four items reduced SSPS performance by 21%, ten items reduced performance by 17%, and twenty items reduced performance by 9%.

Both optimizations would be beneficial in threaded code, but the structure of the resulting server would be isomorphic to the StagedServer version, with a thread bound to each process or performing event lookups on a

subset of the data structure, and a queue in front of each process to accumulate a cohort.

## 6 Related Work

The advantages and disadvantages of threads and processes are widely known [5]. More recently, several papers have investigated alternative server architectures. Chankhunthod et al. described the Harvest web cache, which uses an event-driven, reactive architecture to invoke computation at transitions in its state-machine control logic [13]. The system, like StagedServer, uses non-blocking I/O; careful avoidance of page faults; and a non-blocking, non-preemptive scheduling policy [7, 26]. Pai proposed a four-fold characterization of server architectures: multi-process, multi-threaded, single-process event driven, and asymmetric multi-process event driven [26]. These alternatives are orthogonal to the task scheduling policy, and as the discussion in Section 2 illustrates, cohort scheduling could increase their locality. Pai's favored event-driven programming style offers many opportunities for cohort scheduling, since event handlers partition a computation into distinct, easily identifiable subcomputations with clear operation boundaries. On the other hand, ad-hoc event systems offer no obvious way to group handlers that belong in the same cohort or to associate data with operations. Section 3 describes staged computation, a programming model that provides a programmer with control over the computation in a cohort.

Welsh recently described the SEDA system, which is similar to the staged computation model [29]. SEDA, unlike StagedServer, does not use explicit cohort scheduling, but instead uses stages as an architecture for structuring event-driven servers. His performance results are similar for I/O intensive server applications.

Blackwell used blocked layer processing to improve the instruction locality of a TCP/IP stack [10]. He noted that the TCP/IP code was larger than the MIPS R2000 instruction cache, so that when the protocol stack processed a packet completely, no code from the lower protocol layers remained in cache for the next packet. His solution was to process several packets together at each layer. The modified stack had a lower cache miss rate and reduced processing latency. Blackwell related his approach to blocked matrix computations [19], but his focus was instruction locality. Cohort scheduling, whose genesis predates Blackwell, is a more general scheduling policy and system architecture, which is applicable when a computation is not as cleanly partitionable as a network stack. Moreover, cohort scheduling improves data, not just instruction, locality and reduces synchronization as well.

A stage is similar in some respects to an object in an object-based language, in that it provides both local state and operations to manipulate it. The two differ because objects are, for the most part, passive and their methods are synchronous—though asynchronous object models exist. Many object-oriented languages, such as Java [17], integrate threads and synchronization, but the active entity remains a thread synchronously run a method on a passive object. By contrast, in staged computation, a stage is asked to perform an operation, but is given the autonomy to decide how and when to actually execute the work. This decoupling of request and response is valuable because it enables a stage to control its concurrency and to adopt an efficient scheduling policy, such as cohort scheduling.

Stages are similar in some respects to Agha's Actors [3]. Both start with a model of asynchronous communication between autonomous entities. Actors have no internal concurrency and do not give entities control over their scheduling, but instead presume a reactive model in which an Actor responds to a message by invoking a computation. Stages, because of the internal concurrency and scheduling autonomy, are better suited to cohort scheduling. Actors are, in turn, an instance of dataflow, a more general computing model [23, 24]. Stages also can be viewed as an instance of dataflow computation.

Cilk is language based on a provably efficient scheduling policy [11]. The language is thread, not object, based, but it shares some characteristics with stages. In both, once started, a computation is not pre-empted. While running, a computation can spawn off other tasks, which return their results by invoking a continuation. However, Cilk's work stealing scheduling policy does not implement cohort scheduling, nor is it under program control. Recent work, however, has improved the data locality of work stealing scheduling algorithms [1].

JAWS is an object-oriented framework for writing web servers [18]. It consists of a collection of design patterns, which can be used to construct servers adapted to a particular operating system by selecting an appropriate concurrency mechanism (processes or threads), creating a thread pool, reducing synchronization, caching files, using scatter-gather I/O, or employing various http and TCP-specific optimizations. StagedServer is a simpler library that provides a programming model that directly enhances program locality and performance.

An earlier version of this work was published as a short, extended abstract [21].

## 7 Conclusion

Servers are commonly structured as a collection of parallel tasks, each of which executes all the code necessary to process a request. Threads, processes, or event handlers underlie the software architecture of most servers. Unfortunately, this software architecture can interact poorly with modern processors, whose performance depends on mechanisms—caches, TLBs, and branch predictors—that exploit program locality to bridge the increasing processor-memory performance gap. Servers have little inherent locality. A thread typically runs for a short and unpredictable amount of time and is followed by an unrelated thread, with its own working set. Moreover, servers interact frequently with the operating system, which has a large and disruptive working set. The poor processor performance of servers is a natural consequence of their threaded architecture.

As a remedy, we propose cohort scheduling, which increases server locality by consecutively executing related operations from different server requests. Running similar code on a processor increases instruction and data locality, which aids hardware mechanisms, such as cache and branch predictors. Moreover, this architecture naturally issues operating system requests in batches, which reduces the system's disruption.

This paper also describes the staged computation programming model, which supports cohort scheduling by providing an abstraction for grouping related operations and mechanisms through which a program can implement cohort scheduling. This approach has been implemented in the StagedServer library. In a series of tests using a web server and publish-subscribe server, the StagedServer code performed better than threaded code, with a lower level of cache misses and instruction stalls and better performance under heavy load.

## Acknowledgements

This work has on going for a long time, and countless people have provided invaluable insights and feedback. This list is incomplete; and we apologize for omissions. Rick Vicik made important contributions to the idea of cohort scheduling and the early implementations. Jim Gray has been a ceaseless supporter and advocate of this work. Kevin Zatloukal helped write the web server and run many early experiments. Trishul Chilimbi, Jim Gray, Vinod Grover, Mark Hill, Murali Krishnan, Paul Larson, Milo Martin, Ron Murray, Luke McDowell, Scott McFarling, Simon Peyton-Jones, Mike Smith, and Ben Zorn provided many helpful questions and comments. The referees and shepherd, Carla Ellis, provided many helpful comments.

## References

- [1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe, "The Data Locality of Work Stealing," in Proceedings of the Twelfth ACM Symposium on Parallel Algorithms and Architectures (SPAA). Bar Harbor, ME, July 2000.
- [2] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur, "Cooperative Tasking without Manual Stack Management," in Proceedings of the 2002 USENIX Annual Technical Conference. Monterey, CA, June 2002.
- [3] Gul A. Agha, ACTORS: A Model of Concurrent Computation in Distributed Systems. Cambridge, MA: MIT Press, 1988.
- [4] Anastassia G. Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?," in Proceedings of 25th International Conference on Very Large Data Bases. Edinburgh, Scotland: Morgan Kaufmann, September 1999, pp. 266-277.
- [5] Thomas E. Anderson, "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors," IEEE Transactions on Parallel and Distributed Systems, vol. 1, num. 1, pp. 6-16, 1990.
- [6] Andrew W. Appel, *Compiling with Continuations*. Cambridge University Press, 1992.
- [7] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul, "Better Operating System Features for Faster Network Servers," in Proceedings of the Workshop on Internet Server Performance. Madison, WI, June 1998.
- [8] Paul Barford and Mark Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," in Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems. Madison, WI, June 1998, pp. 151-160.
- [9] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion, "Memory System Characterization of Commercial Workloads," in Proceedings of the 25th Annual International Symposium on Computer Architecture. Barcelona, Spain, June 1998, pp. 3-14.
- [10] Trevor Blackwell, "Speeding up Protocols for Small Messages," in Proceedings of the ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. Palo Alto, CA, August 1996, pp. 85-95.
- [11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou, "Cilk: An Efficient Multithreaded Runtime System," Journal of Parallel and Distributed Computing, vol. 37, num. 1, pp. 55-69, 1996.
- [12] Satish Chandra, Bradley Richards, and James R. Larus, "Teapot: A Domain-Specific Language for Writing Cache Coherence Protocols," IEEE Transactions on Software Engineering, vol. 25, num. 3, pp. 317-333, 1999.
- [13] Anawat Chankhunthod, Peter Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell, "A Hierarchical Internet Object Cache," in Proceedings of the USENIX 1996 Annual Technical Conference. San Diego, CA, January 1996.
- [14] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems," in Proceedings of the Thirteenth ACM Symposium on Operating System Principles. Pacific Grove, CA, October 1991, pp. 122-136.
- [15] Edmund M. Clarke Jr., Oma Grumberg, and Doron A. Peled, *Model Checking*. Cambridge, MA: MIT Press, 1999.
- [16] Françoise Fabret, H. Arno Jacobsen, François Llirbat, Joao Pereira, Kenneth A. Ross, and Dennis Shasha, "Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems," in Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data and Symposium on Principles of Database Systems. Santa Barbara, CA, May 2001, pp. 115-126.
- [17] James Gosling, Bill Joy, and Guy Steele, *The Java Language Specification*. Addison Wesley, 1996.
- [18] James Hu and Douglas C. Schmidt, "JAWS: A Framework for High-performance Web Servers," in Domain-Specific Application Frameworks: Frameworks Experience By Industry, M. Fayad and R. Johnson, Eds.: John Wiley & Sons, October 1999.
- [19] F. Irigoin and R. Troilet, "Supernode Partitioning," in Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages. San Diego, CA, January 1988, pp. 319-329.
- [20] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker, "Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads," in Proceedings of the 25th Annual International Symposium on Computer Architecture. Barcelona, Spain, June 1998, pp. 15-26.
- [21] James R. Larus and Michael Parkes, "Using Cohort Scheduling to Enhance Server Performance (Extended Abstract)," in Proceedings of the Workshop on Optimization of Middleware and Distributed Systems. Snowbird, UT, June 2001, pp. 182-187.
- [22] James R. Larus, Sriram K. Rajamani, and Jakob Rehof, "Behavioral Types for Structured Asynchronous Programming," Microsoft Research, Redmond, WA, May 2001.
- [23] Edward A. Lee and Thomas M. Parks, "Dataflow Process Networks," Proceedings of the IEEE, vol. 83, num. 5, pp. 773-799, 1995.
- [24] Walid A. Najjar, Edward A. Lee, and Guang R. Gao, "Advances in the Dataflow Computation Model," Parallel Computing, vol. 25:1907-1929, 1999.
- [25] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa, "Executing Parallel Programs with Synchronization Bottlenecks Efficiently," in Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA '99). Sendai, Japan: World Scientific, July 1999, pp. 182-204.
- [26] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel, "Flash: An Efficient and Portable Web Server," in Proceedings of the 1999 USENIX Annual Technical Conference. Monterey, CA, June 1999, pp. 199-212.
- [27] David A. Patterson and John L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2 ed. Morgan Kaufmann, 1996.
- [28] Sharon Perl and Richard L. Sites, "Studies of Windows NT Performance using Dynamic Execution Traces," in Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI). Seattle, WA, October 1997, pp. 169-183.
- [29] Matt Welsh, David Culler, and Eric Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," in Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01). Alberta, Canada, October 2001, pp. 230-243.
- [30] Michael J. Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.

# EtE: Passive End-to-End Internet Service Performance Monitoring \*

Yun Fu<sup>†</sup>, Ludmila Cherkasova<sup>‡</sup>, Wenting Tang<sup>‡</sup>, and Amin Vahdat<sup>†</sup>

<sup>†</sup>Dept of Computer Science, Duke University  
Durham, NC 27708, USA  
email: {fu,vahdat}@cs.duke.edu

<sup>‡</sup>Hewlett-Packard Laboratories  
1501 Page Mill Road, Palo Alto, CA 94303, USA  
email: {lucy\_cherkasova, wenting\_tang}@hp.com

**Abstract.** *This paper presents, EtE monitor, a novel approach to measuring web site performance. Our system passively collects packet traces from a server site to determine service performance characteristics. We introduce a two-pass heuristic method and a statistical filtering mechanism to accurately reconstruct different client page accesses and to measure performance characteristics integrated across all client accesses. Relative to existing approaches, EtE monitor offers the following benefits: i) a breakdown between the network and server overhead of retrieving a web page, ii) longitudinal information for all client accesses, not just the subset probed by a third party, iii) characteristics of accesses that are aborted by clients, and iv) quantification of the benefits of network and browser caches on server performance. Our initial implementation and performance analysis across two sample sites confirm the utility of our approach.*

## 1 Introduction

Today, Internet services are delivering a large array of business, government, and personal services. Similarly, mission critical operations, related to scientific instrumentation, military operations, and health services, are making increasing use of the Internet for delivering information and distributed coordination. However, the best effort nature of Internet data delivery, changing client and network connectivity characteristics, and the highly complex architectures of modern Internet services make it very difficult to understand the performance characteristics of Internet services. In a competitive landscape, such understanding is critical to continually evolving and engineering Internet services to match changing demand levels and client populations.

\*This work was originated and largely completed while Y. Fu worked at HPLabs during the summer 2001 and supported in part by research grant from HP. A. Vahdat and Y. Fu are supported in part by the National Science Foundation (EIA-9972879). A. Vahdat is also supported by an NSF CAREER award (CCR-9984328).

Currently, there are two popular techniques for benchmarking the performance of Internet services. The first approach, active probing [13, 17, 23, 19], uses machines from fixed points in the Internet to periodically request one or more URLs from a target web service, record end-to-end performance characteristics, and report a time-varying summary back to the web service. The second approach, web page instrumentation [8, 10, 2, 20], associates code (e.g., JavaScript) with target web pages. The code, after being downloaded into the client browser, tracks the download time for individual objects and reports performance characteristics back to the web site.

In this paper, we present a novel approach to measuring web site performance called EtE monitor. Our system passively collects network packet traces from the server site to enable either offline or online analysis of system performance characteristics. Using two-pass heuristics and statistical filtering mechanisms, we are able to accurately reconstruct individual page composition without parsing HTML files or obtaining out-of-band information about changing site characteristics. Relative to existing techniques, EtE monitor offers a number of benefits:

- Our system can determine the breakdown between the server and network overhead associated with retrieving a web page. This information is necessary to understand where performance optimizations should be directed, for instance to improve server-side performance or to leverage existing content distribution networks (CDNs) to improve network locality.
- EtE monitor tracks all accesses to web pages for a given service. Many existing techniques are typically restricted to a few probes per hour to URLs that are pre-determined to be popular. Our approach is much more agile to changing client access patterns. What real clients are accessing determines the performance that EtE monitor eval-

uates. Finally, given the Zipf popularity of service web pages [1], our approach is able to track the characteristics of the heavy tail that often makes up a large overall portion of web site accesses.

- Given information on all client accesses, clustering techniques [15] can be utilized to determine network performance characteristics by network region or autonomous system. System administrators can use this information to determine which content distribution networks to partner with (depending on their points of presence) or to determine multi-homing strategies with particular ISPs.
- EtE monitor captures information on page requests that are manually aborted by the client, either because of unsatisfactory web site performance or specific client browsing patterns (e.g., clicking on a link before a page has completed the download process). Existing techniques cannot model user interactions in the case of active probing or miss important aspects of web site performance such as TCP connection establishment in the case of web page instrumentation.
- Finally, EtE monitor is able to determine the actual benefits of both browser and network caches. By learning the likely composition of individual web pages, our system can determine when certain embedded objects of a web page are not requested and conclude that those objects were retrieved from some cache in the network.

This paper presents the architecture and implementation of our prototype EtE monitor. It also highlights the benefits of our approach through an evaluation of the performance of two sample network services using EtE monitor. Overall, we believe that detailed performance information will enable network services to dynamically react to changing access patterns and system characteristics to best match client QoS expectations. Depending on the architecture of the system, a front end “Layer-7” switch [18] could redirect requests for particular objects to a smaller or larger set of back-end machines based on observed performance summaries. Similarly, performance characteristics across multiple services being served from a single hosting center can be used to allocate resources to competing services to, for example, maximize aggregate throughput or to maintain higher-level service level agreements [4]. Sites may also use performance information to dynamically adjust system consistency [25] or content fidelity [3] with the goal of meeting target levels of performance.

The rest of this paper is organized as follows. In the next section, we survey existing techniques and products and discuss their merits and drawbacks. Section 3 outlines the EtE monitor architecture, with additional details in Sections 4-6. In Section 7, we present the results

of two performance studies, which have been performed to test and validate EtE monitor and its approach. Section 8 presents two specially designed experiments to validate the accuracy of EtE monitor performance measurements and its page access reconstruction power. We discuss the limitations of the proposed technique in Section 9 and present our conclusions and future work in Section 10.

**Acknowledgments:** Both the tool and the study would not have been possible without generous help of our HP colleagues: Mike Rodriguez, Steve Yonkaitis, Guy Mathews, Annabelle Eseo, Peter Haddad, Bob Husted, Norm Follett, Don Reab, and Vincent Rabiller. Their help is highly appreciated. Our special thanks to Claude Villerman who helped to identify and to correct a subtle bug for dynamic page reconstruction. We would like to thank the anonymous referees for useful remarks and insightful questions, and our shepherd Jason Nieh for constructive suggestions to improve the content and presentation of the paper.

## 2 Related Work

A number of companies use active probing techniques to offer measurement and testing services today, including Keynote [13], NetMechanic [17], Software Research [23], and Porivo Technologies [19]. Their solutions are based on periodic polling of web services using a set of geographically distributed, synthetic clients. In general, only a few pages or operations can typically be tested, potentially reflecting only a fraction of all user's experience. Further, active probing techniques cannot typically capture the potential benefits of browser and network caches, in some sense reflecting “worst case” performance. From another perspective, active probes come from a different set of machines than those that actually access the service. Thus, there may not always be correlation in the performance/reliability reported by the service and that experienced by end users. Finally, it is more difficult to determine the breakdown between network and server-side performance using active probing, making it more difficult for customers to determine where best to place their optimization efforts.

Another popular approach is to embed instrumentation code with web pages to record access times and report statistics back to the server. For instance, WTO (Web Transaction Observer) from HP OpenView suite [8] uses JavaScript to implement this functionality. With additional web server instrumentation and cookie techniques, this product can record the server processing time for a request, enabling a breakdown between server and network processing time. A number of other products and proposals [10, 2, 20] employ similar techniques. Relative to our approach, web page instrumentation can also capture end-to-end performance information from real clients, except connection establishment times (po-



tentially an important aspect of overall performance). Further, this approach requires additional server-side instrumentation and dedicated resources to actively collect performance reports from clients.

There have been some earlier attempts to passively estimate the response time observed by clients from network level information. SPAND [21, 22] determines network characteristics by making shared, passive measurements from a collection of hosts and uses this information for server selection, i.e. for routing client requests to the server with the best observed response time in a geographically distributed web server cluster.

### 3 EtE Monitor Architecture

EtE monitor consists of four program modules shown in Figure 1:

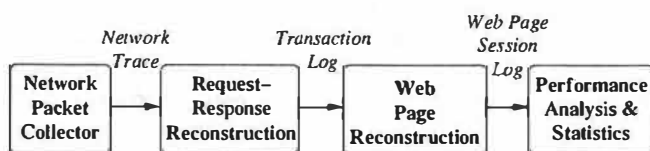


Figure 1: EtE Monitor Architecture.

1. The *Network Packet Collector* module collects the network packets using *tcpdump*[24] and records them to a *Network Trace*, enabling offline analysis.
2. In the *Request-Response Reconstruction* module, EtE monitor reconstructs all TCP connections from the *Network Trace* and extracts HTTP transactions (a request with the corresponding response) from the payload. EtE monitor does not consider encrypted connections whose content cannot be analyzed. After obtaining the HTTP transactions, the monitor stores some HTTP header lines and other related information in the *Transaction log* for future processing (excluding the HTTP payload). To rebuild HTTP transactions from TCP-level traces, we use a methodology proposed by Feldmann [7] and described in more detail and extended to work with persistent HTTP connections by Krishnamurthy and Rexford [14].
3. The *Web Page Reconstruction* module is responsible for grouping underlying physical object retrievals together into logical web pages and stores them in the *Web Page Session Log*.
4. Finally, the *Performance Analysis and Statistics* module summarizes a variety of performance characteristics integrated across all client accesses.

EtE monitor can be deployed in several different ways. First, it can be installed on a web server as a *software*

*component* to monitor web transactions on a particular server. However, our software would then compete with the web server for CPU cycles and I/O bandwidth (as quantified in Section 7). Another solution is to place EtE monitor as an independent *network appliance* at a point on the network where it can capture all HTTP transactions for a web server. If a web site consists of multiple web servers, EtE monitor should be placed at the common entrance and exit of all web servers. If a web site is supported by geographically distributed web servers, such a common point may not exist. Nevertheless, distributed web servers typically use “sticky connections”, i.e., once the client has established a connection with a web server, the subsequent client requests are sent to the same server. In this case, EtE monitor can still be used to capture a flow of transactions to a particular geographic site.

### 4 Request-Response Reconstruction Module

As described above, the Request-Response Reconstruction module reconstructs all observed TCP connections. The TCP connections are rebuilt from the *Network Trace* using client IP addresses, client port numbers, and request (response) TCP sequence numbers. Within the payload of the rebuilt TCP connections, HTTP transactions can be delimited as defined by the HTTP protocol. Meanwhile, the timestamps, sequence numbers and acknowledged sequence numbers for HTTP requests can be recorded for later matching with the corresponding HTTP responses.

When a client clicks a hypertext link to retrieve a particular web page, the browser first establishes a TCP connection with the web server by sending a SYN packet. If the server is ready to process the request, it accepts the connection by sending back a second SYN packet acknowledging the client’s SYN<sup>1</sup>. At this point, the client is ready to send HTTP requests to retrieve the HTML file and all embedded objects. For each request, we are concerned with the timestamps for the first byte and the last byte of the request since they delimit the request transfer time and the beginning of server processing. We are similarly concerned with the timestamps of the beginning and the end of the corresponding HTTP response.

EtE monitor detects aborted connections by observing either a RST packet sent by an HTTP client to explicitly indicate an aborted connection or by a FIN/ACK

<sup>1</sup>Whenever EtE monitor detects a SYN packet, it considers the packet as a new connection iff it cannot find a SYN packet with the same source port number from the same IP address. A retransmitted SYN packet is not considered as a newly established connection. However, if a SYN packet is dropped, e.g. by intermediate routers, there is no way to detect the dropped SYN packet on server side.

packet sent by the client where the acknowledged sequence number is less than the observed maximum sequence number sent from the server. After reconstructing the HTTP transactions (a request and the corresponding response), the monitor records the HTTP header lines in the *Transaction Log* and discards the actual body of the HTTP response.

Each entry in the log includes a number of fields: (1) a unique flow ID for the TCP connection, (2) the client's IP address, (3) the requested URL, (4) the content type, (5) the *referer* field, (6) the *via* field, (7) whether the request was aborted, (8) the number of packets resent during the connection (potentially an indication of the presence of network congestion), (9) the size and timestamps of the request and response. Some fields in the entry are used to rebuild web pages, while other fields can be used to measure end-to-end performance.

An alternative way to collect most of the fields of the *Transaction Log* entry is to extend web server functionality. Apache, Netscape and IIS all have appropriate APIs. Most of the fields in the *Transaction Log* can be extracted via server instrumentation. This approach has some merits: 1) since a web server deals directly with request-response processing, the reconstruction of TCP connections becomes unnecessary; 2) it can handle encrypted connections.

However, the primary drawback of this approach is that web servers must be modified in an application specific manner. Our approach is independent of any particular server technology. On the other hand, instrumentation solutions cannot obtain network level information, such as the connection setup time and the resent packets, which can be observed by EtE monitor.

## 5 Page Reconstruction Module

To measure the client perceived end-to-end response time for retrieving a web page, one needs to identify the objects that are embedded in a particular web page and to measure the response time for the client requests retrieving these embedded objects from the web server. Although we can determine some embedded objects of a web page by parsing the HTML for the "container object", some embedded objects cannot be easily discovered through static parsing. For example, JavaScript is used in web pages to retrieve additional objects. Without executing the JavaScript, it may be difficult to discover the identity of such objects.

Automatically, determining the content of a page requires a technique to delimit individual page accesses. One recent study [6] uses an estimate of client think time as the delimiter between two pages. While this method is simple and useful, it may be inaccurate in some important cases. For example, consider the case where a client opens two web pages from one server at the same time. Here, the requests for the two different web pages

interleave each other without any think time between them. Another case is when the interval between the requests for objects within one page may be too long to be distinguishable from think time (perhaps because of the network conditions).

Different from previous work, our methodology uses heuristics to determine the objects composing a web page, and applies statistics to adjust the results. EtE uses the HTTP *referer* field as a major "clue" to group objects into a web page. The *referer* field specifies the URL from which the requested URL was obtained. Thus, all requests for the embedded objects in an HTML file are recommended to set the *referer* fields to the URL of the HTML file. However, since the *referer* fields are set by client browsers, not all browsers set the fields. To solve this, EtE monitor first builds a *Knowledge Base* from those requests with *referer* fields, and uses more aggressive heuristics to group the requests without *referer* fields based on the *Knowledge Base* information.

Subsection 5.1 outlines *Knowledge Base* construction of web page objects. Subsection 5.2 presents the algorithm and technique to group the requests in web page accesses using *Knowledge Base* information and a set of additional heuristics. Subsection 5.3 introduces a statistical analysis to identify valid page access patterns and to filter out incorrectly constructed accesses.

### 5.1 Building a Knowledge Base of Web Page Objects

The goal of this step is to reconstruct a special subset of web page accesses, which we use to build a *Knowledge Base* about web pages and the objects composing them. Before grouping HTTP transactions into web pages, EtE monitor first sorts all transactions from the *Transaction Log* using the timestamps for the beginning of the requests in increasing time order. Thus, the requests for the embedded objects of a web page must follow the request for the corresponding HTML file of the page. When grouping objects into web pages (here and in the next subsection), we consider only transactions with *successful* responses, i.e. with status code 200 in the responses.

The next step is to scan the sorted transaction log and group objects into web page accesses. Not all the transactions are useful for the *Knowledge Base* construction process. During this step, some of the *Transaction Log* entries are excluded from our current consideration:

- Content types that are known not to contain embedded objects are excluded from the knowledge base, e.g., *application/postscript*, *application/x-tar*, *application/pdf*, *application/zip* and *text/plain*. For the rest of the paper, we call them *independent, single page* objects.
- If the *referer* field of a transaction is not set and its

content type is not *text/html*, EtE monitor excludes it from further consideration.

To group the rest of the transactions into web page accesses, we use the following fields from the entries in the *Transaction Log*: the request URL, the request *referer* field, the response *content type*, and the client IP address. EtE monitor stores the web page access information into a hash table, the *Client Access Table* depicted in Figure 2, which maps a client's IP address to a *Web Page Table* containing the web pages accessed by the client. Each entry in the *Web Page Table* is a web page access, and composed of the URLs of HTML files and the embedded objects. Notice that EtE monitor makes no distinction between statically and dynamically generated HTML files. We consider embedded HTML pages, e.g. framed web pages, as separate web pages.

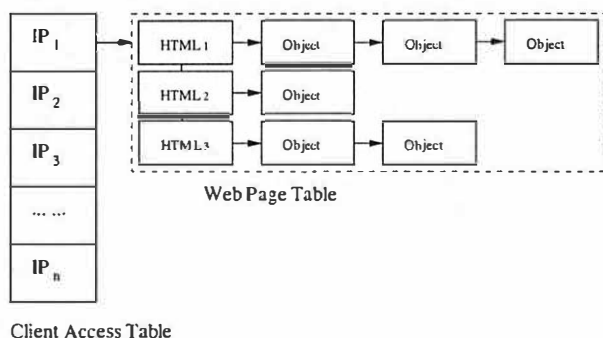


Figure 2: *Client Access Table*.

When processing an entry of the *Transaction Log*, EtE monitor first locates the *Web Page Table* for the client's IP in the *Client Access Table*. Then, EtE monitor handles the transaction according to its content type:

1. If the content type is *text/html*, EtE monitor treats it as the beginning of a web page and creates a new web page entry in the *Web Page Table*.

2. For other content types, EtE monitor attempts to insert the URL of the requested object into the web page that contains it according to its *referer* field. If the referred HTML file is already present in the *Web Page Table*, EtE monitor appends this object at the end of the entry. If the referred HTML file does not exist in the client's *Web Page Table*, it means that the client may have retrieved a cached copy of the object from somewhere else between the client and the web server. In this case, EtE monitor first creates a new web page entry in the *Web Page Table* for the referred HTML file. Then it appends the considered object to this page.

From the *Client Access Table*, EtE monitor determines the *content template* of any given web page as a combined set of all the objects that appear in all the access patterns for this web page. Thus, EtE monitor scans the *Client Access Table* and creates a new hash table, as shown in Figure 3, which is used as a *Knowledge*

*Base* to group the accesses for the same web pages from other client's browsers that do not set the *referer* fields.

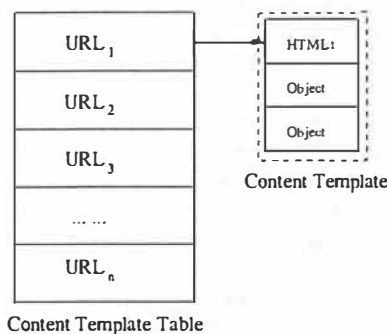


Figure 3: *Knowledge Base* of web pages.

## 5.2 Reconstruction of Web Page Accesses

With the help of the *Knowledge Base*, EtE monitor processes the entire *Transaction Log* again. This time, EtE monitor does not exclude the entries without *referer* fields. Using data structures similar to those introduced in Section 5.1, EtE monitor scans the sorted *Transaction Log* and creates a new *Client Access Table* to store all accesses as depicted in Figure 2. For each transaction, EtE monitor locates the *Web Page Table* for the client's IP in the *Client Access Table*. Then, EtE monitor handles the transaction depending on the content type:

1. If the content type is *text/html*, EtE monitor creates a new web page entry in the *Web Page Table*.
2. If a transaction is an independent, single page object, EtE monitor marks it as individual page without any embedded objects and allocates a new web page entry in the *Web Page Table*.
3. For other content types that can be embedded in a web page, EtE monitor attempts to insert it into the web page that contains it.

- If the *referer* field is set for this transaction, EtE monitor attempts to locate the referred page in the following way. If the referred HTML file is in an existing page entry in the *Web Page Table*, EtE monitor appends the object at the end of the entry. If the referred HTML file does not exist in the client's *Web Page Table*, EtE monitor first creates a new web page entry in the table for the referred page and marks it as *nonexistent*. Then it appends the object to this page. If the *referer* field is not set for this transaction, EtE monitor uses the following policies. With the help of the *Knowledge Base*, EtE monitor checks each page entry in the *Web Page Table* from the latest to earliest. If the *Knowledge Base* contains the *content template* for the checked

page and the considered object does not belong to it, EtE monitor skips the entry and checks the next one until a page containing the object is found. If such an entry is found, EtE monitor appends the object to the end of the web page.

- If none of the web page entries in the *Web Page Table* contains the object based on the *Knowledge Base*, EtE monitor searches in the client's *Web Page Table* for a web page accessed via the same flow ID as this object. If there is such a web page, EtE monitor appends the object to the page.
- Otherwise, if there are any accessed web pages in the table, EtE monitor appends the object to the latest accessed one.

If none of the above policies can be applied, EtE monitor drops the request. Obviously, the above heuristics may introduce some mistakes. Thus, EtE monitor also adopts a *configurable think time threshold* to delimit web pages. If the time gap between the object and the tail of the web page that it tries to append to is larger than the threshold, EtE monitor skips the considered object. In this paper, we adopt a configurable think time threshold of 4 sec.

### 5.3 Identifying Valid Accesses Using Statistical Analysis of Access Patterns

Although the above two-pass process can effectively provide accurate web page access reconstruction in most cases, there could still be some accesses grouped incorrectly. To filter out such accesses, we must better approximate the actual content of a web page.

All the accesses to a web page usually exhibit a set of different access patterns. For example, an access pattern can contain all the objects of a web page, while other patterns may contain a subset of them (e.g., because some objects were retrieved from a browser or network caches). We assume the same access patterns of those incorrectly grouped accesses should rarely appear repeatedly. Thus, we can use the following statistical analysis on access patterns to determine the actual content of web pages and exclude the incorrectly grouped accesses.

First, from the *Client Access Table* created in Subsection 5.2, EtE monitor collects all possible access patterns for a given web page and identifies the *probable content template* of the web page as the combined set of all objects that appear in all the accesses for this page. Table 1 shows an example of a *probable content template*. EtE monitor assigns an index for each object. The column *URL* lists the URLs of the objects that appear in the access patterns for the web page. The column *Frequency* shows the frequency of an object in the set of all web page accesses. In Table 1, the indices are sorted by the

occurrence frequencies of the objects. The column *Ratio* is the percentage of the object's accesses in the total accesses for the page.

Index	URL	Frequency	Ratio (%)
1	/index.html	2937	95.51
2	/img1.gif	689	22.41
3	/img2.gif	641	20.85
4	/log1.gif	1	0.03
5	/log2.gif	1	0.03

Table 1: Web page *probable content template*. There are 3075 accesses for this page.

Sometimes, a web page may be pointed to by several URLs. For example, <http://www.hpl.hp.com> and <http://www.hpl.hp.com/index.html> both point to the same page. Before computing the statistics of the access patterns, EtE monitor attempts to merge the accesses for the same web page with different URL expressions. EtE monitor uses the *probable content* templates of these URLs to determine whether they indicate the same web page. If the *probable content* templates of two pages only differ due to the objects with small percentage of accesses (less than 1%, which means these objects might have been grouped by mistake), then EtE monitor ignores this difference and merges the URLs.

Based on the *probable content template* of a web page, EtE monitor uses the indices of objects in the table to describe the access patterns for the web page. Table 2 demonstrates a set of different access patterns for the web page in Table 1. Each row in the table is an access pattern. The column *Object Indices* shows the indices of the objects accessed in a pattern. The columns *Frequency* and *Ratio* are the number of accesses and the proportion of the pattern in the total number of all the accesses for the web page. For example, pattern 1 is a pattern in which only the object *index.html* is accessed. It is the most popular access pattern for this web page: 2271 accesses out of the total 3075 accesses represent this pattern. In pattern 2, the objects *index.html*, *img1.gif* and *img2.gif* are accessed.

Pattern	Object Indices	Frequency	Ratio (%)
1	1	2271	73.85
2	1,2,3	475	15.45
3	1,2	113	3.67
4	1,3	76	2.47
5	2,3	51	1.66
6	2	49	1.59
7	3	38	1.24
8	1,2,4	1	0.03
9	1,3,5	1	0.03

Table 2: Web page access patterns.

With the statistics of access patterns, EtE monitor further attempts to estimate the *true content template* of web pages, which excludes the mistakenly grouped access patterns. Intuitively, the proportion of these invalid

access patterns cannot be high. Thus, EtE monitor uses a configurable ratio threshold to exclude the invalid patterns (in this paper, we use 1% as a configurable ratio threshold). If the ratio of a pattern is below the threshold, EtE does not consider it as a valid pattern. In the above example, patterns 8 and 9 are not considered as valid access patterns. Only the objects found in the valid access patterns are considered as the embedded objects in a given web page. Objects 1, 2, and 3 define the *true content template* of the web page shown in Table 3. Based on the *true content templates*, EtE monitor filters out all the invalid accesses in a *Client Access Table*, and records the correctly constructed page accesses in the *Web Page Session Log*, which can be used to evaluate the end-to-end response performance.

Index	URL
1	/index.html
2	/img1.gif
3	/img2.gif

Table 3: Web page *true content template*.

## 6 Metrics to Measure Web Service Performance

In this section, we introduce a set of metrics and the ways to compute them in order to measure a web service efficiency. These metrics can be categorized as:

- metrics approximating the end-to-end response time observed by the client for a web page download. Additionally, we provide a means to calculate the breakdown between server processing and networking portions of overall response time.
- metrics evaluating the caching efficiency for a given web page by computing the server file hit ratio and server byte hit ratio for the web page.
- metrics relating the end-to-end performance of aborted web pages to the QoS.

### 6.1 Response Time Metrics

We use the following functions to denote the critical timestamps for connection *conn* and request *r*:

- $t_{syn}(conn)$ : time when the first SYN packet from the client is received for establishing the connection *conn*;
- $t_{req}^{start}(r)$ : time when the first byte of the request *r* is received ;
- $t_{req}^{end}(r)$ : time when the last byte of the request *r* is received;

- $t_{resp}^{start}(r)$ : time when the first byte of the response for *r* is sent;
- $t_{resp}^{end}(r)$ : time when the last byte of the response for *r* is sent;
- $t_{ack}^{ack}(r)$ : time when the ACK for the last byte of the response for *r* is received.

Additionally, for a web page *P*, we have the following variables:

- *N* - the number of distinct connections used to retrieve the objects in the web page *P*;
- $r_1^k, \dots, r_{n_k}^k$  - the requests for the objects retrieved through the connection *conn<sub>k</sub>* (*k* = 1, ..., *N*), and ordered accordingly to the time when these requests were received, i.e.,

$$t_{req}^{end}(r_1^k) \leq t_{req}^{end}(r_2^k) \leq \dots \leq t_{req}^{end}(r_{n_k}^k).$$

The extended version of HTTP 1.0 and later version HTTP 1.1 [9] introduce the concepts of *persistent connections* and *pipelining*. Persistent connections enable reuse of a single TCP connection for multiple object retrievals from the same IP address. Pipelining allows a client to make a series of requests on a persistent connection without waiting for the previous response to complete (the server must, however, return the responses in the same order as the requests are sent).

We consider the requests  $r_1^k, \dots, r_{n_k}^k$  to belong to the same *pipelining group* (denoted as *PipeGr* =  $\{r_i^k, \dots, r_n^k\}$ ) if for any *j* such that  $i \leq j - 1 < j \leq n$ ,  $t_{req}^{start}(r_j^k) \leq t_{resp}^{end}(r_{j-1}^k)$ .

Thus for all the requests on the same connection *conn<sub>k</sub>*:  $r_1^k, \dots, r_{n_k}^k$ , we define the maximum pipelining groups in such a way that they do not intersect, e.g.,

$$\underbrace{r_1^k, \dots, r_i^k}_{PipeGr_1}, \underbrace{r_{i+1}^k}_{PipeGr_2}, \dots, \underbrace{r_{n_k}^k}_{PipeGr_l}$$

For each of the pipelining groups, we define three portions of response time: total response time (*Total*), network-related portion (*Network*), and lower-bound estimate of the server processing time (*Server*).

Let us consider the following example. For convenience, let us denote  $PipeGr_1 = \{r_1^k, \dots, r_i^k\}$ .

Then

$$Total(PipeGr_1) = t_{resp}^{end}(r_i^k) - t_{req}^{start}(r_1^k),$$

$$Network(PipeGr_1) = \sum_{j=1}^i (t_{resp}^{end}(r_j^k) - t_{resp}^{start}(r_j^k)),$$

$$Server(PipeGr_1) = Total(PipeGr_1) - Network(PipeGr_1).$$

If no pipelining exists, a pipelining group only consists of one request. In this case, the computed server time represents precisely the server processing time for a given



request-response pair. If a connection adopts pipelining, the “real” server processing time might be larger than the computed server time because it can partially overlap the network transfer time, and it is difficult to estimate the exact server processing time from the packet-level information. However, we are still interested to estimate the “non-overlapping” server processing time as this is the portion of the server time on a critical path of overall end-to-end response time. Thus, we use as an estimate the lower-bound server processing time, which is explicitly exposed in the overall end-to-end response.

If connection  $conn_k$  is a newly established connection to retrieve a web page, we observe additional connection setup time:

$$Setup(conn_k) = t_{req}^{start}(r_1^k) - t_{syn}(conn_k)^2,$$

otherwise the setup time is 0. Additionally, we define  $t_{start}^{start}(conn_k) = t_{syn}(conn_k)$  for a newly established connection, otherwise,  $t_{start}^{start}(conn_k) = t_{req}^{start}(r_1^k)$ .

Similarly, we define the breakdown for a given connection  $conn_k$ :

$$Total(conn_k) = Setup(conn_k) + t_{resp}^{end}(r_{n_k}^k) - t_{req}^{start}(r_1^k),$$

$$Network(conn_k) = Setup(conn_k) + \sum_{j=1}^l Network(PipeGr_j),$$

$$Server(conn_k) = \sum_{j=1}^l Server(PipeGr_j).$$

Now, we define similar latencies for a given page  $P$ :

$$Total(P) = \max_{j \leq N} t_{resp}^{end}(r_{n_j}^j) - \min_{j \leq N} t_{req}^{start}(conn_j),$$

$$CumNetwork(P) = \sum_{j=1}^N Network(conn_j),$$

$$CumServer(P) = \sum_{j=1}^N Server(conn_j).$$

For the rest of the paper, we will use the term *EtE time* interchangeably with *Total(P)* time.

All the above formulae use  $t_{resp}^{end}(r)$  to calculate response time. An alternative way is to use as the end of a transaction the time  $t_{resp}^{ack}(r)$  when the ACK for the last byte of the response is received by a server. Figure 4 shows an example of a simplified scenario where a 1-object page is downloaded by the client: it shows the communication protocol for connection setup between the client and the server as well as the set of major timestamps collected by the EtE monitor on the server side. The connection setup time measured on the server side is the time between the client SYN packet and the first byte of the client request. This represents a close approximation for the original client setup time (we present

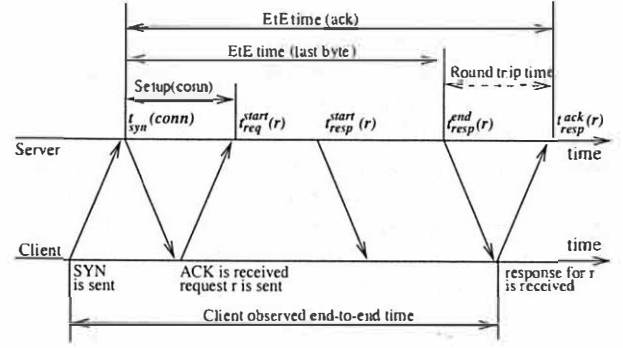


Figure 4: An example of a 1-object page download by the client: major timestamps collected by the EtE monitor on the server side.

more detail on this point in Section 8 when reporting our validation experiments).

If the ACK for the last byte of the client response is not delayed or lost,  $t_{resp}^{ack}(r)$  is a more accurate approximation of the end-to-end response time observed by the client: it “compensates” for the latency of the first client SYN packet that is not measured on the server side. The difference between the two methods, i.e. *EtE time (last byte)* and *EtE time (ack)*, is only a round trip time, which is on the scale of milliseconds. Since the overall response time is on the scale of seconds, we consider this deviation an acceptably close approximation. However, to avoid the problems with delayed or lost ACKs, EtE monitor determines the end of a transaction as the time when the last byte of a response is sent by a server.

Metrics introduced in this section account for packet retransmission. However, if the retransmission happens on connection establishment (i.e. due to dropped SYNs), EtE monitor cannot account for this.

The functions *CumNetwork(P)* and *CumServer(P)* give the sum of all the network-related and server processing portions of the response time over all connections used to retrieve the web page. However, the connections can be opened concurrently by the browser. To evaluate the concurrency impact, we introduce the page concurrency coefficient *ConcurrencyCof(P)*:

$$ConcurrencyCof(P) = \frac{\sum_{j=1}^N Total(conn_j)}{Total(P)}.$$

Using page concurrency coefficient, we finally compute the network-related and the service-related portions of response time for a particular page  $P$ :

$$Network(P) = CumNetwork(P) / ConcurrencyCof(P),$$

$$Server(P) = CumServer(P) / ConcurrencyCof(P).$$

EtE monitor can distinguish the requests sent to a web server from clients behind proxies by checking the

<sup>2</sup>The connection setup time as measured by EtE monitor does not include dropped SYNs, as discussed earlier in Section 4.

HTTP *via* fields. If a client page access is handled via the same proxy (which is typically the case, especially when persistent connections are used), EtE monitor provides correct measurements for end-to-end response time and other metrics, as well as provides interesting statistics on the percentage of client requests coming from proxies. Clearly, this percentage is an approximation, since not all the proxies set the *via* fields in their requests. Finally, EtE monitor can only measure the response time to a proxy instead of the actual client behind it.

## 6.2 Metrics Evaluating the Web Service Caching Efficiency

Real clients of a web service may benefit from the presence of network and browser caches, which can significantly reduce their perceived response time. However, none of the existing performance measurement techniques provide any information on the impact of caches on web services: what percentage of the files and bytes are delivered from the server comparing with the total files and bytes required for delivering the web service. This impact can only be partially evaluated from web server logs by checking response status code 304, whose corresponding requests are sent by the network caches to validate whether the cached object has been modified. If the status code 304 is set, the cached object is not expired and need not be retrieved again.

To evaluate the caching efficiency of a web service, we introduce two metrics: *server file hit ratio* and *server byte hit ratio* for each web page.

For a web page  $P$ , assume the objects composing the page are  $O_1, \dots, O_n$ . Let  $Size(O_i)$  denote the size of object  $O_i$  in bytes. Then we define  $NumFiles(P) = n$  and  $Size(P) = \sum_{j=1}^n Size(O_j)$ .

Additionally, for each access  $P_{access}^i$  of the page  $P$ , assume the objects retrieved in the access are  $O_1^i, \dots, O_{k_i}^i$ , we define  $NumFiles(P_{access}^i) = k_i$  and  $Size(P_{access}^i) = \sum_{j=1}^{k_i} Size(O_j^i)$ . First, we define *file hit ratio* and *byte hit ratio* for each page access in the following way:

$$FileHitRatio(P_{access}^i) = NumFiles(P_{access}^i) / NumFiles(P),$$

$$ByteHitRatio(P_{access}^i) = Size(P_{access}^i) / Size(P).$$

Let  $P_{access}^1, \dots, P_{access}^N$  be all the accesses to the page  $P$  during the observed time interval. Then

$$ServerFileHitRatio(P) = \frac{1}{N} \sum_{k=1}^N FileHitRatio(P_{access}^k),$$

$$ServerByteHitRatio(P) = \frac{1}{N} \sum_{k=1}^N ByteHitRatio(P_{access}^k).$$

The lower numbers for *server file hit ratio* and *server byte hit ratio* indicate the higher caching efficiency for the web service, i.e., more files and bytes are served from network and client browser caches.

## 6.3 Aborted Pages and QoS

User-perceived QoS is another important metric to consider in EtE monitor. One way to measure the QoS of a web service is to measure the frequency of aborted connections. However, such simplistic interpretation of aborted connections and web server QoS has several drawbacks. First, a client can interrupt HTTP transactions by clicking the browser's "stop" or "reload" button while a web page is downloading, or clicking a displayed link before the page is completely downloaded. Thus, only a subset of aborted connections are relevant to poor web site QoS or poor networking conditions, while other aborted connections are caused by client-specific browsing patterns. On the other hand, a web page can be retrieved through multiple connections. A client's browser-level interruption can cause all the currently open connections to be aborted. Thus, the number of aborted page accesses more accurately reflects client satisfaction than the number of aborted connections.

For aborted pages, we distinguish the subset of pages  $\Pi_{bad}$  with the response time higher than the given threshold  $X_{EtE}$  (in our case,  $X_{EtE} = 6$  sec). Only these pages might be reflective of the bad quality downloads. While a simple deterministic cut off point cannot truly capture a particular client's expectation for site performance, the current industrial *ad hoc* quality goal is to deliver pages within 6 sec [12]. We thus attribute aborted pages that have not crossed the 6 sec threshold to individual client browsing patterns. The next step is to distinguish the reasons leading to poor response time: whether it is due to network or server-related performance problems, or both.

## 7 Case Studies

In this section, we present two simple case studies to illustrate the benefits of EtE monitor in assessing web site performance. The first site is the HP Labs external site (*HPL Site*), <http://www.hpl.hp.com>. Static web pages comprise most of this site's content. We measured performance of this site for a month, from July 12, 2001 to August 11, 2001. The second site is a support site for a popular HP product family, which we call *Support Site*. It uses JavaServer Pages [11] technology for dynamic page generation. The architecture of this site is based on a geographically distributed web server cluster with Cisco Distributed Director [5] for load balancing, using "sticky connections". We measure the site performance for 2 weeks, from October 11, 2001 to October 25, 2001.

Table 4 summarizes the two site's performance *at-a-glance* during the measured period using the two most frequently accessed pages at each site. The average end-to-end response time of client accesses to these pages reflects good overall performance. However in the case of

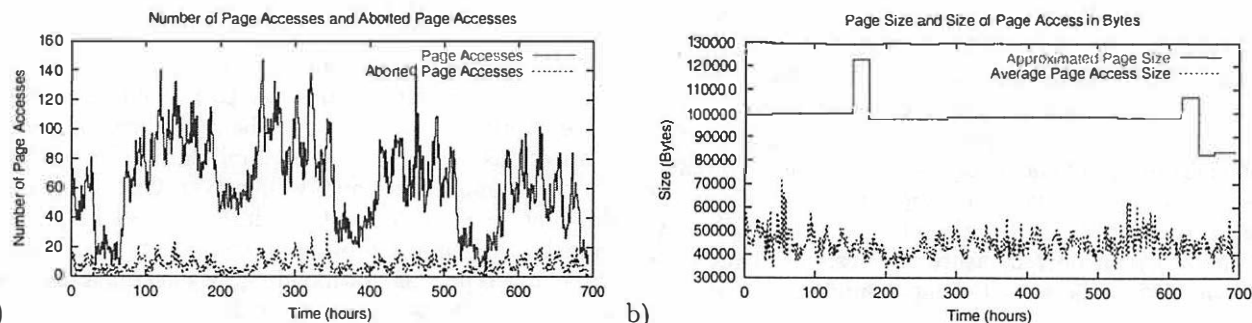


Figure 5: HPL site during a month: a) Number of all and aborted accesses to *index.html*; b) Approximated page size and average access size to *index.html*.

Metrics	HPL <i>url1</i>	HPL <i>url2</i>	Support <i>url1</i>	Support <i>url2</i>
EtE time	3.5 sec	3.9 sec	2.6 sec	3.3 sec
% of accesses above 6 sec	8.2%	8.3%	1.8%	2.2%
% of aborted accesses above 6 sec	1.3%	2.8%	0.1%	0.2%
% of accesses from clients-proxies	16.8%	19.8%	11.2%	11.7%
EtE time from clients-proxies	4.2 sec	3 sec	4.5 sec	3 sec
Network-vs-Server ratio in EtE time	99.6%	99.7%	96.3%	93.5%
Page size	99 KB	60.9 KB	127 KB	100 KB
Server file hit ratio	38.5%	58%	22.9%	28.6%
Server byte hit ratio	44.5%	63.2%	52.8%	44.6%
Number of objects	4	2	32	32
Number of connections	1.6	1	6.5	9.1

Table 4: At-a-Glance statistics for *www.hpl.hp.com* and *support* site during the measured period.

HPL, a sizeable percentage of accesses take more than 6 sec to complete (8.2%-8.3%), with a portion leading to aborted accesses (1.3%-2.8%). The Support site had better overall response time with a much smaller percentage of accesses above 6 sec (1.8%-2.2%), and a correspondingly smaller percentage of accesses aborted due to high response time (0.1%-0.2%). Overall, the pages from both sites are comparable in size. However, the two pages from the HPL site have a small number of objects per page (4 and 2 correspondingly), while the Support site pages are composed of 32 different objects. Page composition influences the number of client connections required to retrieve the page content. Additionally, statistics show that network and browser caches help to deliver a significant amount of page objects: in the case of the Support site, only 22.9%-28.6% of the 32 objects are retrieved from the server, accounting for 44.6%-52.8% of the bytes in the requested pages. As discussed earlier, the Support site content is generated using dynamic pages, which could potentially lead to a higher ratio of server processing time in the overall re-

sponse time. But in general, the network transfer time dominates the performance for both sites, ranging from 93.5% for the Support site to 99.7% for the HPL site.

Given the above summary, we now present more detailed information from our site measurements. For the HPL site, the two most popular pages during the observed period were *index.html* and a page in the news section describing the Itanium chip (we call it *itanium.html*).

Figure 5 a) shows the number of page accesses to *index.html*, as well as the number of aborted page accesses during the measured period. The graph clearly reflects weekly access patterns to the site.

Figure 5 b) reflects the *approximate page size*, as reconstructed by EtE monitor. We use this data to additionally validate the page reconstruction process. While debugging the tool, we manually compare the content of the 20 most frequently accessed pages reconstructed by EtE monitor against the actual web pages: the EtE monitor page reconstruction accuracy for popular pages is very high, practically 100%. Figure 5 b) allows us to "see" the results of this reconstruction process over the period of the study. In the beginning, it is a straight line exactly coinciding with the actual page size. At hour mark 153, it jumps and returns to a next straight line interval at the 175 hour mark. As we verified, the page has been partially modified during this time interval. The EtE monitor "picked" both the old and the modified page images, since they both occurred during the same day interval and represented a significant fraction of accesses. However, the next day, the *Knowledge Base* was "renewed" and had only the modified page information. The second "jump" of this line corresponds to the next modification of the page. The gap can be tightened, depending on the time interval EtE monitor is set to process. The other line in Figure 5 b) shows the average page access size, reflecting the server byte hit ratio of approximately 44%.

To characterize the reasons leading to the aborted web pages, we present analysis of the aborted accesses to *index.html* page for 3 days in August (since the monthly graph looks very "busy" on an hourly scale). Figure 6 a)

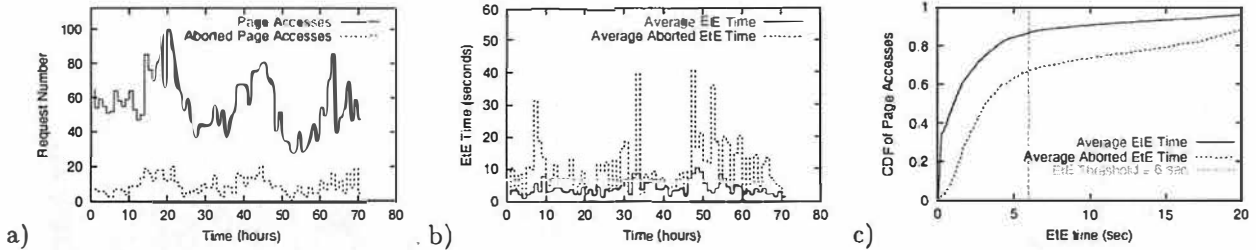


Figure 6: HPL site during 3 days: a) Number of all and aborted accesses to *index.html*; b) End-to-end response times for accesses to *index.html*; c) CDF of all and aborted accesses to *index.html* sorted by the response time in increasing order.

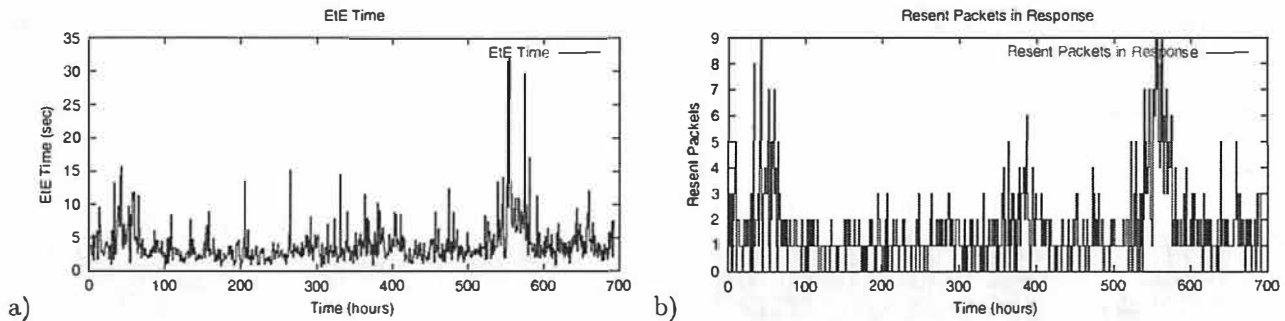


Figure 7: HPL site during a month: a) end-to-end response times for accesses to *index.html*; b) number of resent packets in response.

shows the number of all the requests and the aborted requests to *index.html* page during this interval. The number of aborted accesses (662) accounts for 16.4% of the total number of requests (4028).

Figure 6 b) shows the average end-to-end response time measured by EtE monitor for *index.html* and the average end-to-end response time for the aborted accesses to *index.html* on an hourly scale. The end-to-end response time for *index.html* page, averaged across all the page accesses, is 3.978 sec, while the average end-to-end response time of the aborted page accesses is 9.21 sec.

Figure 6 c) shows a cumulative distribution of all accesses and aborted accesses to *index.html* sorted by the end-to-end response time in increasing order. The vertical line on the graph shows the threshold of 6 sec that corresponds to an acceptable end-to-end response time. Figure 6 c) shows that 68% of the aborted accesses demonstrate end-to-end response times below 6 sec. This means that only 32% of all the aborted accesses, which in turn account for 5% of all accesses to the page, observe high end-to-end response time. The next step is to distinguish the reasons leading to a poor response time: whether it is due to network or server performance problems, or both. For all the aborted pages with high response time, the network portion of the response time dominates the overall response time (98%-99% of the total). Thus, we can conclude that any performance problems are likely not server-related but rather due to congestion in the network (though it is unclear whether the congestion is at the edge or the core of the network).

Figure 7 a) shows the end-to-end response time for ac-

cesses to *index.html* on an hourly scale during a month. In spite of good average response time reported in at-a-glance table, hourly averages reflect significant variation in response times. This graph helps to stress the advantages of EtE monitor and reflects the shortcomings of active probing techniques that measure page performance only a few times per hour: the collected test numbers could vary significantly from a site's instantaneous performance characteristics.

Figure 7 b) shows the number of resent packets in the response stream to clients. There are three pronounced "humps" with an increased number of resent packets. Typically, resent packets reflect network congestion or the existence of some network-related bottlenecks. Interestingly enough, such periods correspond to weekends when the overall traffic is one order of magnitude lower than weekdays (as reflected in Figure 5 a)). The explanation for this phenomenon is that during weekends the client population of the site "changes" significantly: most of the clients access the site from home using modems or other low-bandwidth connections. This leads to a higher observed end-to-end response time and an increase in the number of resent packets (i.e., TCP is likely to cause drops more often when probing for the appropriate congestion window over a low-bandwidth link). These results again stress the unique capabilities of EtE monitor to extract appropriate information from network packets, and reflect another shortcoming of active probing techniques that use a fixed number of artificial clients with rather good network connections to the Internet. For site designers, it is important to

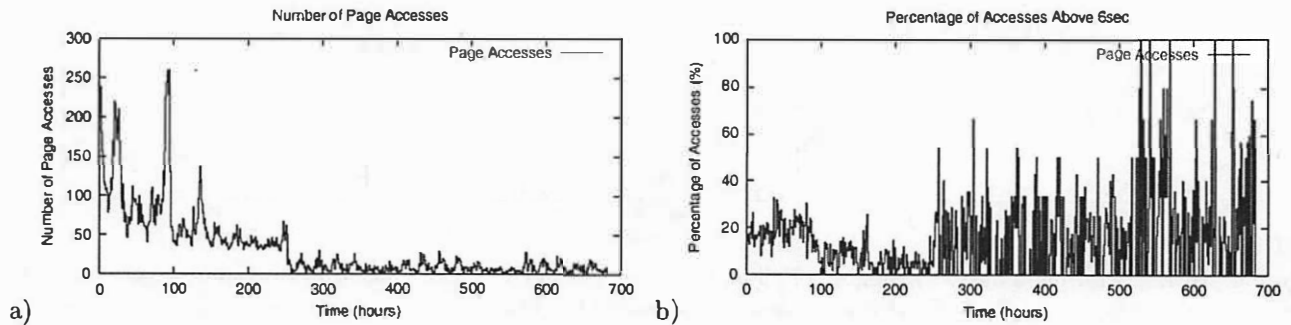


Figure 8: HPL site during a month: a) number of all accesses to *itanium.html*; b) percentage of accesses with end-to-end response time above 6 sec.

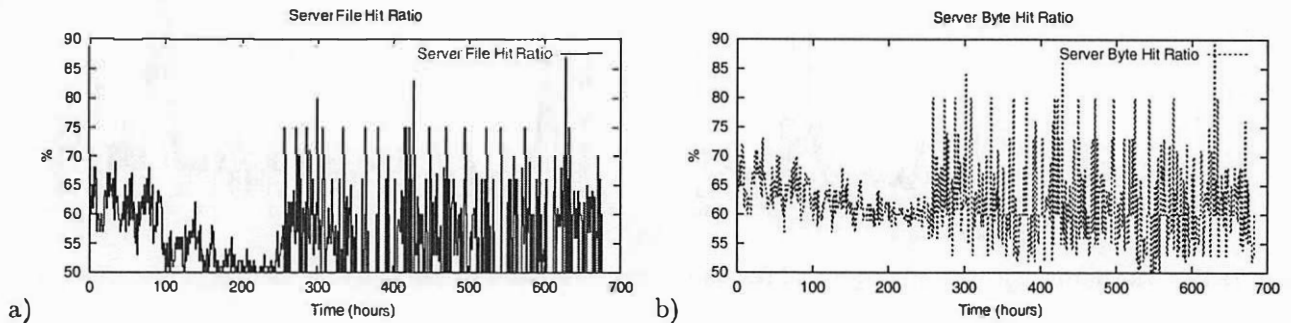


Figure 9: HPL site: a) server file hit ratio for *itanium.html*; b) server byte hit ratio for *itanium.html*.

understand the actual client population and their end-to-end response time and the “quality” of the response. For instance, when large population of clients have limited bandwidth parameters, the site designers should consider making the pages and their objects “lighter weight”.

Figure 8 a) shows the number of page accesses to *itanium.html*. When we started our measurement of the HPL site, the *itanium.html* page was the most popular page, “beating” the popularity of the main *index.html* page. However, ten days later, this news article started to get “colder”, and the page got to the seventh place by popularity.

Figure 8 b) shows the percentage of accesses with end-to-end response time above 6 sec. The percentage of high response time jumps significantly when the page becomes “colder”. The reason behind this phenomenon is shown in Figure 9, which plots the server file hit and byte hit ratio. When the page became less popular, the number of objects and the corresponding bytes retrieved from the server increased significantly. This reflects that fewer network caches store the objects as the page becomes less popular, forcing clients to retrieve them from the origin server.

Figure 8 b) and Figure 9 explicitly demonstrate the network caching impact on end-to-end response time. When the caching efficiency of a page is higher (i.e., more page objects are cached by network and browser caches), the response time measured by EtE monitor is

lower. Again, active probing techniques cannot measure (or account for) the page caching efficiency to reflect the “true” end-to-end response time observed by the actual clients.

We now switch to the analysis of the Support site. We will only highlight some new observations specific to this site. Figure 10 a) shows the average end-to-end response time as measured by EtE monitor when downloading the site main page. This site uses JavaServer Pages technology for dynamic generation of the content. Since dynamic pages are typically more “compute intensive,” it has a corresponding reflection in higher server-side processing fraction in overall response time. Figure 10 b) shows the network-server time ratio in the overall response time. It is higher compared to the network-server ratio for static pages from the HPL site. One interesting detail is that the response time spike around the 127 hour mark has a corresponding spike in increased server processing time, indicating some server-side problems at this point. The combination of data provided by EtE monitor can help service providers to better understand site-related performance problems.

The Support site pages are composed of a large number of embedded images. Two most popular site pages, which account for almost 50% of all the page accesses, consist of 32 objects. The caching efficiency for the site is very high: only 8-9 objects are typically retrieved from the server, while the other objects are served from network and browser caches. The site server is running



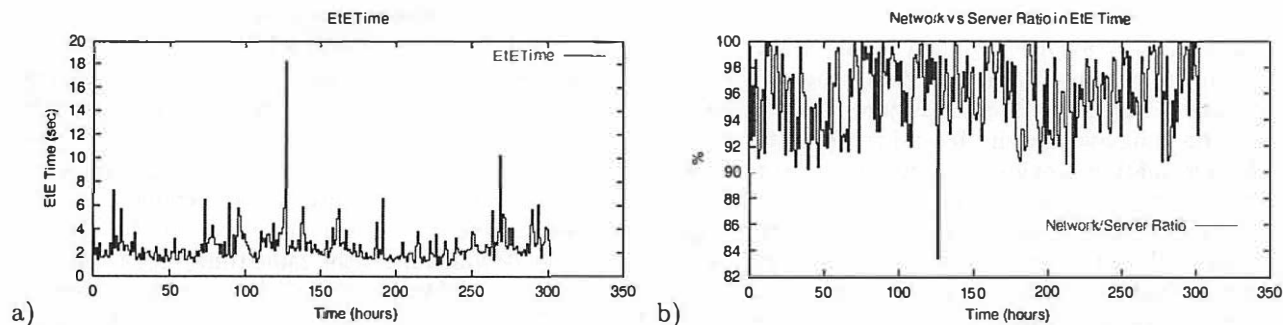


Figure 10: Support site during 2 weeks: a) end-to-end response time for accesses to a main page; b) network-server time ratio for the main page.

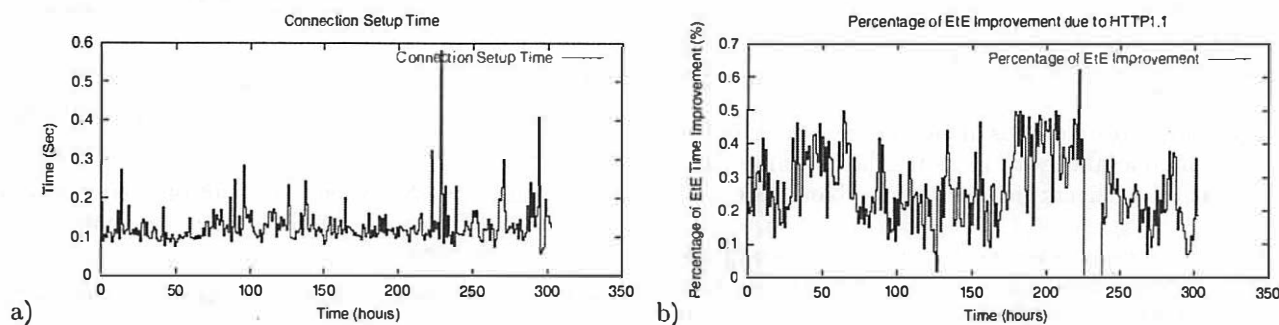


Figure 11: Support site during 2 weeks: a) connection setup time for the main page; b) an estimated percentage of end-to-end response time improvement if the server runs HTTP1.1.

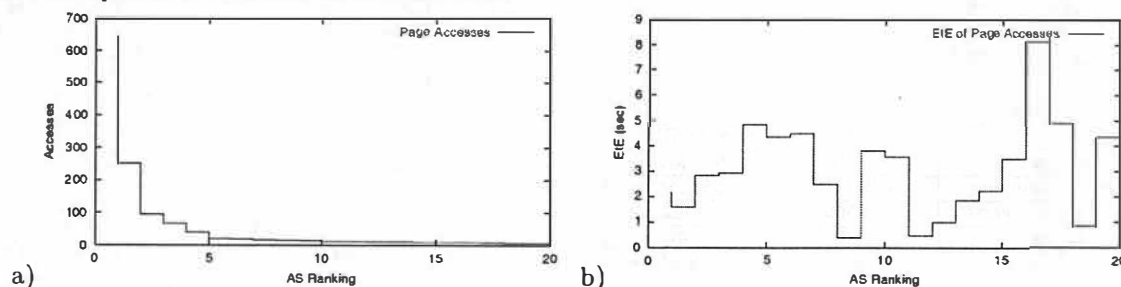


Figure 12: Support site: daily analysis of 20 ASes with largest client clusters: a) number of different clients accessing the main page; b) corresponding end-to-end response time per AS.

HTTP 1.0 server. Thus typical clients used 7-9 connections to retrieve 8-9 objects. The *ConcurrencyCoef* (see Section 6), which reflects the overlap portion of the latency between different connections for this page, was very low, around 1.038 (in fact, this is true for the site pages in general). This indicates that the efficiency of most of these connections is almost equal to sequential retrievals through a single persistent connection.

Figure 11 a) shows the connection setup time measured by EtE monitor. We perform a simple computation: how much of the end-to-end response time observed by current clients can be improved if the site server would run an HTTP 1.1 server, allowing clients to use just two persistent connections to retrieve the corresponding objects from the site? In other words, how much of the response time can be improved by eliminating unnecessary connection setup time?

Figure 11 b) shows the estimated percentage of end-to-end response time improvement available from running an HTTP 1.1 server. On average, during the observed interval, the response time improvement for *url1* is around 20% (2.6 sec is decreased to 2.1 sec), and for *url2* is around 32% (3.3 sec is decreased to 2.2 sec).

Figure 11 b) reveals an unexpected “gap” between 230-240 hour marks, when there was “no improvement” due to HTTP 1.1. More careful analysis shows that during this period, all the accesses retrieved only a basic HTML page using 1 connection, without consequent image retrievals. The other pages during the same interval have a similar pattern. It looks like the image directory was not accessible on the server. Thus, EtE monitor, by exposing the abnormal access patterns, can help service providers get additional insight into service related problems.

EtE monitor also provides the information about the client clustering by associating them with various ASes (Autonomous Systems). Figure 12 a) shows the 20 largest client clusters by ASes. Figure 12 b) reflects the corresponding average end-to-end response time per AS. The information provides a useful quantitative view on response times to the major client clusters. It can be used for efficient site design when the geographically distributed web cluster is needed to improve site performance. Similarly, such information can be used to make appropriate decisions on specific content distribution networks and wide-area replication strategies given a particular service's client population.

The ability of EtE monitor to reflect a site performance for different ASes (and groups of IP addresses) happens to be a very attractive feature for service providers. When service providers have special SLA-contracts with certain groups of customers, EtE monitor provides a unique ability to measure the response time observed by those clients and validate QoS for those contracts.

Finally, we present a few performance numbers to reflect the execution time of EtE monitor when processing data for the HPL and Support sites. The tests are run on a 550Mhz HP C3600 workstation with 512 MB of RAM. Table 5 presents the amount of data and the execution time for processing 10,000,000 TCP Packets.

Duration, Size, and Execution Time	HPL site	Support site
Duration of data collection	3 days	1 day
Collected data size	7.2 GB	8.94 GB
Transaction Log size	35 MB	9.6 MB
Entries in Transaction Log	616,663	157,200
Reconstructed page accesses	90,569	8,642
Reconstructed pages	5,821	845
EtE Execution Time	12 min 44 sec	17 min 41 sec

Table 5: EtE monitor performance measurements.

The performance of reconstruction module performance depends on the complexity of the web page composition. For example, the Support site has a much higher percentage of embedded objects per page than the HPLabs pages. This "higher complexity" of the reconstruction process is reflected by the higher EtE monitor processing time for the Support site (17 min 41 sec) compared to the processing time for the HPLabs site (12 min 44 sec). The amount of incoming and outgoing packets of a web server farm that an EtE monitor can handle also depends on the rate at which *tcpdump* can capture packets and the traffic of the web site.

## 8 Validation Experiments

We performed two groups of experiments to validate the accuracy of EtE monitor performance measurements and its page access reconstruction power.

In the first experiment, we used two remote clients residing at Duke University and Michigan State University to issue a sequence of 40 requests to retrieve a designated web page from HPLabs external web site, which consists of an HTML file and 7 embedded images. The total page size is 175 Kbytes. To issue these requests, we use *httperf*[16], a tool which measures the connection setup time and the end-to-end time observed by the client for a full page download. At the same time, an EtE monitor measures the performance of HPLabs external web site. From EtE monitor measurements, we filter the statistics about the designated client accesses. Additionally, in EtE monitor, we compute the end-to-end time using two slightly different approaches from those discussed in Section 6.1:

- EtE time (*last byte*): where the *end* of a transaction is the time when the last byte of the response is sent by a server;
- EtE time (*ACK*): where the *end* of a transaction is the time when the ACK for the last byte of the response is received.

Table 6 summarizes the results of this experiment (the measurements are given in *sec*):

Client	<i>httperf</i>		<i>EtE monitor</i>		
	Conn Setup	Resp. time	Conn Setup	EtE time ( <i>last byte</i> )	EtE time ( <i>ACK</i> )
Michigan	0.074	1.027	0.088	0.953	1.026
Duke	0.102	1.38	0.117	1.28	1.38

Table 6: Experimental results validating the accuracy of EtE monitor performance measurements.

The connection setup time reported by EtE monitor is slightly higher (14-15 ms) than the actual setup time measured by *httperf*, since it includes the time to not only establish a TCP connection but also receive the first byte of a request. The EtE time (*ACK*) coincides with the actual measured response time observed by the client. The EtE time (*last byte*) is slightly lower than the actual response time by exactly a round trip delay (the connection setup time measured by *httperf* represents the round trip time for each client, accounting for 74-102 ms). These measurements correctly reflect our expectations for EtE monitor accuracy (see Section 6.1). Thus, we have some confidence that EtE monitor accurately approximates the actual response time observed by the client.

The second experiment was performed to evaluate the reconstruction power of EtE monitor. The EtE monitor with its two-pass heuristic method actively uses the *referrer* field to reconstruct the page composition and to build a *Knowledge Base* about the web pages and objects composing them. This information is used during the second pass to more accurately group the requests into page accesses. The question to answer is: how dependent are the reconstruction results on the existence

of *referer* field information. If the *referer* field is not set in most of the requests, how is the EtE monitor reconstruction process affected? How is the reconstruction process affected by accesses generated by proxies?

To answer these questions, we performed the following experiment. To reduce the incorrectness introduced by proxies, we first filtered the requests with *via* fields, which are issued by proxies, from the original *Transaction Logs* for the both sites. These requests constitute 24% of total requests for the HPL site and 1.1% of total requests for the Support site. We call these logs *filtered logs*. Further, we mask the *referer* fields of all transactions in the *filtered logs* to study the correctness of reconstruction. We call these modified logs *masked logs*, which do not contain any *referer* fields. We notice that the requests with *referer* fields constitute 56% of the total requests for the HPL site and 69% for the Support site in the *filtered logs*. Then, EtE monitor processes the *filtered logs* and *masked logs*. Table 7 summarizes the results of this experiment.

Metrics	HPL <i>url1</i>	HPL <i>url2</i>	Support <i>url1</i>	Support <i>url2</i>
Reconstructed page accesses ( <i>filtered logs</i> )	36,402	17,562	17,601	11,310
EtE time ( <i>filtered logs</i> )	3.3 sec	4.1 sec	2.4 sec	3.3 sec
Reconstructed page accesses ( <i>masked logs</i> )	33,735	14,727	15,401	8,890
EtE time ( <i>masked logs</i> )	3.2 sec	4.1 sec	2.3 sec	3.6 sec

Table 7: Experimental results validating the accuracy of EtE monitor reconstruction process for HPL and Support sites.

The results of *masked logs* in Table 7 show that EtE monitor does a good job of page access reconstruction even when the requests do not have any *referer* fields. However, with the knowledge introduced by the *referer fields* in the *filtered logs*, the number of reconstructed page accesses increases by 9-21% for the considered URLs in Table 7. Additionally, we also find that the number of reconstructed accesses increases by 11.2-19.8% for all the considered URLs if EtE monitor processes the original logs without filtering either the *via* fields or the *referer* fields. The difference of EtE time between the two kinds of logs in Table 7 can be explained by the difference of the number of reconstructed accesses. Intuitively, more reconstructed page accesses lead to higher accuracy of estimation. This observation also challenges the accuracy of active probing techniques considering their relatively small sampling sets.

## 9 Limitations

There are a number of limitations to our EtE monitor architecture. Since EtE monitor extracts HTTP transactions by reconstructing TCP connections from captured network packets, it is unable to obtain HTTP information from encrypted connections. Thus, EtE monitor is

not appropriate for sites that encrypt much of their data (e.g., via SSL).

In principle, EtE monitor must capture all traffic entering and exiting a particular site. Thus, our software must typically run on a single web server or a web server cluster with a single entry/exit point where EtE monitor can capture all traffic for this site. If the site “outsources” most of its popular content to CDN-based solutions then EtE monitor can only provide the measurement information about the “rest” of the content, which is delivered from the original site. For sites using CDN-based solutions, the active probing or page instrumentation techniques are more appropriate solutions to measure the site performance. A similar limitation applies to pages with “mixed” content: if a portion of a page (e.g., an embedded image) is served from a remote site, then EtE monitor cannot identify this portion of the page and cannot provide corresponding measurements. In this case, EtE monitor consistently identifies the portion of the page that is stored at the local site, and provides the corresponding measurements and statistics. In many cases, such information is still useful for understanding the performance characteristics of the local site.

The EtE monitor does not capture DNS lookup times. Only active probing techniques are capable of measuring this portion of the response times. Further, for clients behind proxies, EtE monitor can only measure the response times to the proxies instead of to the actual clients.

As discussed in Section 3, the heuristic we use to reconstruct page content may determine incorrect page composition. Although the statistics of access patterns can filter invalid accesses, it works best when the sample size is large enough.

Dynamically generated web pages introduce another issue with our statistical methods. In some cases, there is no consistent *content template* for a dynamic web page if each access consists of different embedded objects (for example, some pages use a rotated set of images or are personalized for client profiles). In this case, there is a danger that metrics such as the *server file hit ratio* and the *server byte hit ratio* introduced in Section 6 may be inaccurate. However, the end-to-end time will be computed correctly for such accesses.

There is an additional problem (typical for server access log analysis of e-commerce sites) about how to aggregate and report the measurement results for dynamic sites where most page accesses are determined by *URLs* with client customized parameters. For example, an e-commerce site could add some client specific parameters to the end of a common URL path. Thus, each access to this logically same URL has a different URL expression. However, service providers may be able to provide the policy to generate these URLs. With the help of the policy description, EtE monitor is still able to aggregate these URLs and measure server performance.

## 10 Conclusion and Future Work

Today, understanding the performance characteristics of Internet services is critical to evolving and engineering Internet services to match changing demand levels, client populations, and global network characteristics. Existing tools for evaluating web service performance typically rely on active probing to a fixed set of URLs or on web page instrumentation that monitors download performance to a client and transmits a summary back to a server. This paper presents, EtE monitor, a novel approach to measuring web site performance. Our system passively collects packet traces from the server site to determine service performance characteristics. We introduce a two-pass heuristic method and a statistical filtering mechanism to accurately reconstruct composition of individual page and performance characteristics integrated across all client accesses.

Relative to existing approaches, EtE monitor offers the following benefits: i) a breakdown between the network and server overhead of retrieving a web page, ii) longitudinal information for all client accesses, not just the subset probed by a third party, iii) characteristics of accesses that are aborted by clients, and iv) quantification of the benefits of network and browser caches on server performance. Our initial implementation and performance analysis across two sample sites confirm the utility of our approach. We are currently investigating the use of our tool to understand the client performance on a per-network region. This analysis can aid in the placement of wide-area replicas or in the choice of an appropriate content distribution network. Finally, our architecture is general to analyzing the performance of multi-tiered web services. For example, application-specific log processing can be used to reconstruct the breakdown of latency across tiers for communication between a load balancing switch and a front end web server, or communication between a web server and the storage tier/database system.

## References

- [1] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching, and Zipf-like Distributions: Evidence, and Implications, In Proceedings of IEEE INFOCOM, March, 1999.
- [2] Candle Corporation: eBusiness Assurance. <http://www.candle.com/>.
- [3] S. Chandra, C. Schlatter Ellis and A. Vahdat. Differentiated Multimedia Web Services Using Quality Aware Transcoding. In Proceedings of IEEE INFOCOM 2000, Tel Aviv, March 2000.
- [4] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP), October, 2001.
- [5] Cisco DistributedDirector., <http://www.cisco.com/>.
- [6] F.D. Smith, F.H. Campos, K. Jeffay, and D. Ott. What TCP/IP Protocol Headers Can Tell Us About the Web. In Proceedings of ACM SIGMETRICS, Cambridge, May, 2001.
- [7] A. Feldmann. BLT: Bi-Layer Tracing of HTTP and TCP/IP. Proceedings of WWW-9, May 2000.
- [8] HP Corporation. OpenView Products: Web Transaction Observer. <http://www.openview.hp.com/>.
- [9] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. RFC 2616, IETF, June 2001. <http://www.w3.org/Protocols/rfc2616/rfc2616.html>.
- [10] IBM Corporation. Tivoli Web Management Solutions, <http://www.tivoli.com/products/demos/twsm.html>.
- [11] JavaServer Pages. <http://java.sun.com/products/jsp/technical.html>.
- [12] T. Keeley. Thin, High Performance Computing over the Internet. Invited talk at Eighth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'2000).
- [13] Keynote Systems, Inc. <http://www.keynote.com>.
- [14] B. Krishnamurthy and J. Rexford. Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement, pp.511-522, Addison Wesley, 2001.
- [15] B. Krishnamurthy and J.Wang, On Network-Aware Clustering of Web Clients. Proceedings of ACM SIGCOMM 2000, August 2000.
- [16] D. Mosberger and T. Jin. Httpperf—A Tool for Measuring Web Server Performance. J. of Performance Evaluation Review, Volume 26, Number 3, December 1998.
- [17] NetMechanic, Inc. <http://www.netmechanics.com>.
- [18] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. 8th International Conference on Architectural Support for Programming Languages and Operating Systems, 1998.
- [19] Porivo Technologies, Inc. <http://www.porivo.com>.
- [20] R. Rajamony, M. Elnozahy. Measuring Client-Perceived Response Times on the WWW. Proceedings of the Third USENIX Symposium on Internet Technologies and Systems (USITS), March 2001, San Francisco.
- [21] S. Seshan, M. Stemm and R. Katz. SPAND: Shared Passive Network Performance Discovery USENIX Symposium on Internet Technologies and Systems, 1997.
- [22] Mark Stemm, Randy Katz, Srinivasan Seshan. A Network Measurement Architecture for Adaptive Applications. Proc. of IEEE INFOCOM, 2000.
- [23] Software Research Inc. <http://www.soft.com>.
- [24] <http://www.tcpcdump.org>.
- [25] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. Proceedings of Operating Systems Design and Implementation (OSDI), October 2000.

# The Performance of Remote Display Mechanisms for Thin-Client Computing

S. Jae Yang, Jason Nieh, Matt Selsky, and Nikhil Tiwari

*Department of Computer Science*

*Columbia University*

{syl80, nieh, selsky, nst8}@cs.columbia.edu

## Abstract

The growing popularity of thin-client systems makes it important to determine the factors that govern the performance of these thin-client architectures. To assess the viability of the thin-client computing model, we measured the performance of six popular thin-client platforms—Citrix MetaFrame, Microsoft Terminal Services, Sun Ray, Tarantella, VNC, and X—running over a wide range of network access bandwidths. We find that thin-client systems can perform well on web and multimedia applications in LAN environments, but the efficiency of the thin-client protocols varies widely. We analyze the differences in the various approaches and explain the impact of the underlying remote display protocols on overall performance. Our results quantify the impact of different approaches in display encoding primitives, display update policies, and display caching and compression techniques across a broad range of thin-client systems.

## 1. Introduction

In the last two decades, the centralized computing model of mainframe computing has shifted to the more distributed model of desktop computing. But as these personal desktop computers become ubiquitous in today's large corporate and academic organizations, the total cost of owning and maintaining them can become unmanageable. In response to this challenge, there is a growing movement to return to a more centralized and easier-to-manage computing strategy. The thin-client computing model is the embodiment of that movement.

The goal of the thin-client model is to centralize computing resources, with all the attendant benefits of easier maintenance and cheaper upgrades, while maintaining the same quality of service for the end user that could be provided by a dedicated workstation. In a thin-client computing environment, end users move from full-featured computers to thin clients, lightweight machines primarily used for display and input and which require less maintenance and less frequent upgrades. Organizations then provide computing services to their end users' thin clients from high-powered servers over a network connection. Server resources can be shared across many users, resulting in more effective utilization of computing hardware.

While thin-client computing is reminiscent of the days of mainframe computing, today's users can no longer be satisfied by dumb terminals that only input and output ASCII text. Thin clients must be able to support graphical computing environments effectively to meet the users' demands. The key mechanism for achieving this is a remote display protocol that enables

graphical displays to be served across a network to a client device, while all application logic is executed on the server. Using such a protocol, the client transmits user input to the server, and the server returns screen updates to the client. For some thin-client systems, no unrecoverable state is stored on the client at all.

Because of the potential cost benefits of thin-client computing, a wide range of thin-client platforms have been developed. Some are designed specifically for use over high-bandwidth local area networks, while others attempt to provide quality service over slow network connections. Some application service providers (ASPs) are even offering thin-client service over wide area networks such as the Internet [3, 21]. The growing popularity of thin-client systems makes it important to analyze their performance, to assess the general feasibility of the thin-client computing model, and to compare various thin-client platforms and determine the factors that govern their performance. However, while many thin-client platforms and protocols have been developed, most of these systems and their protocols are proprietary, and few of the vendors have provided detailed performance measurements for their own products or a cross-platform analysis against other vendors' products.

To assess the viability of the thin-client computing model, we have measured the performance of thin-client computing platforms running over a wide range of network access bandwidths. We have characterized the design choices of underlying remote display technologies and quantified the performance impact of these choices. We considered a range of design choices as exhibited by six of the most popular thin-client



Platform	Display Encoding	Screen Updates	Compression	Client Caching	Client Cache Size	Max Client Display	Transport Protocol
Citrix MetaFrame (ICA)	Low-level graphics	Server-push, lazy	RLE	Glyphs, small bitmaps in memory; large bitmaps on disk	3 MB RAM, Percent of disk (1% default)	8-bit color*	TCP/IP
Microsoft Terminal Services (RDP)	Low-level graphics	Server-push, lazy	RLE	Glyphs, small bitmaps in memory; large bitmaps on disk	1.5 MB RAM, 10 MB disk	8-bit color	TCP/IP
Tarantella (AIP)	Low-level graphics	Server-push, eager or lazy depending on bandwidth, load	Adaptively enabled, RLE and LZW at low bandwidths	Glyphs, pixmaps, files	1024 objects	8-bit color	TCP/IP
AT&T VNC	2D draw primitives	Client-pull, lazy updates between client requests discarded	Hextile (2D RLE)	Only local framebuffer (Copyrect)	N/A	24-bit color	TCP/IP
Sun Ray	2D draw primitives	Server-push, eager	None	Only local framebuffer	N/A	24-bit color	UDP/IP
X	High-level graphics	Server-push, eager	None	Application / toolkit-specific, usually none	N/A	24-bit color	TCP/IP

\* Citrix MetaFrame XP offers the option of 24-bit color depth, but this was not available in time for our experiments.

**Table 1:** Characteristics of thin-client platforms.

platforms in use today: Citrix MetaFrame [5, 14], Microsoft Windows 2000 Terminal Services [6], AT&T Virtual Network Computing (VNC) [22, 32], Tarantella [24, 27], Sun Ray [26, 30], and X [25]. These platforms were chosen for their popularity, performance, and diverse design approaches.

We report the first quantitative measurements to examine the performance of such a broad range of thin-client architectures in various network environments. Because many thin-client systems are closed-source and proprietary, we employed slow-motion benchmarking [37], a novel non-intrusive measurement technique that addresses some of the fundamental difficulties in previous studies of thin-client performance. Our results show that thin-client computing can deliver good performance for web and multimedia applications, but performance varies widely among different thin-client platform designs. Our results show that a simple pixel-based remote display approach can deliver superior performance to more complex thin-client systems that are currently popular. We analyze the differences in the underlying mechanisms of various thin-client platforms and explain their impact on overall performance.

This paper is organized as follows. Section 2 details the experimental testbed and methodology we used for our study. Section 3 describes our measurements and performance results. Section 4 discusses some related work. Finally, we present some concluding remarks and directions for future work.

## 2. Experimental Design

The goal of our research was to compare thin-client systems to assess their basic display performance

in various network environments. In our experiments, we used the following six versions of thin-client platforms: Citrix MetaFrame 1.8 for Windows 2000, Windows 2000 Terminal Services, Tarantella Enterprise Express II for Linux, AT&T VNC v3.3.2 for Linux, Sun Ray I for Solaris, and Xfree86 3.3.6 on Linux. In this paper, we also refer to these platforms by their remote display protocols, which are Citrix ICA (Independent Computing Architecture), Microsoft RDP (Remote Desktop Protocol), Tarantella AIP (Adaptive Internet Protocol), VNC, Sun Ray, and X, respectively. As summarized in Table 1, these platforms span a range of differences in the encoding of display primitives, policies for updating the client display, algorithms for compressing screen updates, supported display color depth, and transport protocol used. To evaluate their performance, we designed an experimental testbed and various experiments to exercise each of the thin-client platforms on single-user web-based and multimedia-oriented workloads using slow-motion benchmarking as explained in Section 2.1. Section 2.2 describes the experimental testbed we used. Section 2.3 discusses the application benchmarks used in our experiments.

### 2.1 Measurement Methodology

To provide a more effective method for evaluating thin-client performance, we previously developed slow-motion benchmarking [37]. We developed this benchmarking technique in order to address the inadequacies in conventional benchmarks in measuring thin-client performance. In thin-client systems, the client display is often decoupled from the server-side application execution. In some systems, the screen

updates may be merged or even discarded in order to synchronize the display with the application logic. While these techniques allow the thin server to run the application without being constrained by the slow display update speed, they pose a unique challenge in benchmarking. Standard benchmarks designed for desktop systems cannot be used to provide accurate results when evaluating thin-client systems. Because the benchmark applications are executed on the thin server, independent of the client-side display updates, the benchmarks effectively only measure the server's performance and do not accurately reflect the user's experience at the client-side. A video playback benchmark, for example, would measure the frame rate as rendered on the server, but if many of the frames did not reach the client, the frame rate reported by the benchmark would give an exaggerated view of the system's performance. While internal instrumentation may be an effective solution to this problem, many thin-client products are proprietary and closed-source, making it difficult to instrument them and obtain accurate results. Internal instrumentation can also add intrusive processing overhead.

In slow-motion benchmarking, we use network packet traces to monitor the latency and data transferred between the client and the server, but we alter the benchmark application by inserting delays between the separate visual events, such as web pages or video frames, so that the display update for each event is fully completed on the client before the server begins processing the next one. Then we process the network packet traces and use these gaps of idle time between events to break up the results on a per-event basis. This allows us to obtain the latency and data transferred for each visual event separately. We can then obtain overall results by taking the sum of these per-event results. The amount of the delay inserted depends on the application workload and platform being tested. The necessary length of delay can be determined by monitoring the network traffic and making the delays long enough to achieve a clearly demarcated period between all the visual events where client-server communication drops to the idle level. This ensures that each visual event is discrete and generated completely.

## 2.2 Experimental Testbed

To verify our results in a controlled network environment and to provide a basis for comparison, we constructed an isolated network testbed. Our experimental testbed consisted of seven machines, five of which were active for any given test. The testbed consisted of a network emulator machine, a packet monitor machine, two pairs of thin client/server systems, and a web server used for the web benchmark.

The network emulator machine was a Micron Client Pro PC with two 10/100BaseT NICs running The Cloud [29], a network emulator that we used to adjust the network bandwidth between the client and server. For our experiments, we considered the performance of thin-client systems over a range of network bandwidths, specifically 128 Kbps, 768 Kbps, 1.5 Mbps, 10 Mbps, and 100 Mbps, corresponding roughly to ISDN, DSL, T1, 10BaseT, and 100BaseT, respectively. The packet monitor machine was a Micron Client Pro PC running Etherpeek 4 [33], a network traffic monitor that we used to obtain the measurements for slow-motion benchmarking. To ensure a level playing field, we used the same client/server hardware for all of our tests except when testing the Sun Ray platform, which only runs on Sun machines. The features of each system are summarized in Table 2. As discussed in Section 3, the slower Sun client and server hardware did not affect the lessons derived from our experiments.

Unless otherwise stated, the video resolution of the client was set to 1024x768 with 8-bit color, as this was the lowest common denominator supported by all of the platforms. However, the Sun Ray client was set to 24-bit color, since the Sun Ray display protocol is based on a 24-bit color encoding. By default, compression and memory caching were left on for those platforms that used it, and disk caching was turned off by default in those platforms that supported it. For each thin-client system, we used the server operating system that delivered the best performance for the given system; Terminal Services only runs on Windows. MetaFrame ran best on Windows. Tarantella, VNC, and X ran best on UNIX/Linux, and Sun Ray runs only on Solaris.

## 2.3 Application Benchmarks

To measure the performance of the thin-client platforms, we used two application benchmarks: a web benchmark for measuring web browsing performance, and a video benchmark for measuring video playback performance. The web and video benchmarks were used with the slow-motion benchmarking technique mentioned in Section 2.1 to measure thin-client performance effectively. We describe each of these benchmarks below.

### 2.3.1 Web Benchmark

The web benchmark we used was based on the Web Text Page Load test from the Ziff-Davis i-Bench benchmark suite [10]. We first describe the original i-Bench web benchmark and then discuss how it was modified for our experiments. The original i-Bench web benchmark loads a JavaScript-controlled sequence of 54 web pages from the web benchmark server.

Role / Model	Hardware	OS / Window System	Software
PC Thin Client Micron Client Pro	450 MHz Intel PII 128 MB RAM 14.6 GB Disk 10/100BaseT NIC	MS Win 2000 Professional Caldera OpenLinux 2.4, Xfree86 3.3.6, KDE 1.1.2	Citrix ICA Win32 Client MS RDP5 Client VNC Win32 3.3.3r7 Client SCO Tarantella Win32 Client Netscape Communicator 4.72
Sun Thin Client Sun Ray I	100 MHz Sun uSPARC IIcp 8 MB RAM 10/100BaseT NIC	Sun Ray OS	N/A
Packet Monitor Micron Client Pro	450 MHz Intel PII 128 MB RAM 14.6 GB Disk 10/100BaseT NIC	MS Win 2000 Professional	AG Group's Etherpeek 4
Benchmark Server Micron Client Pro	450 MHz Intel PII 128 MB RAM 14.6 GB Disk 10/100BaseT NIC	MS Win NT 4.0 Server SP6a	Ziff-Davis i-Bench 1.5 MS Internet Information Server
PC Thin-Client Server Micron Client Pro (SPEC95 – 17.2 int, 12.9 fp)	450 MHz Intel PII 128 MB RAM 14.6 GB Disk 2 10/100BaseT NICs	MS Win 2000 Advanced Server Caldera OpenLinux 2.4, Xfree86 3.3.6, KDE 1.1.2	Citrix MetaFrame 1.8 MS Win 2000 Terminal Services AT&T VNC 3.3.3r7 for Win32 SCO Tarantella Express AT&T VNC 3.3.3r2 for Linux Netscape Communicator 4.72
Sun Thin-Client Server Sun Ultra-10 Creator 3D (SPEC95 – 14.2 int, 16.9 fp)	333 MHz UltraSPARC III 384 MB RAM 9 GB Disk 2 10/100BaseT NICs	Sun Solaris 7 Generic 106541-08, OpenWindows 3.6.1, CDE 1.3.5	Sun Ray Server 1.2_10.d Beta Netscape Communicator 4.72
Network Simulator Micron Client Pro	450 MHz Intel PII 128 MB RAM 14.6 GB Disk 2 10/100BaseT NICs	MS Win NT 4.0 Server SP6a	Shunra Software The Cloud 1.1

**Table 2:** Testbed machine configurations.

Normally, as each page downloads, a small script contained in each page starts off the subsequent download. The pages contain both text and bitmap images, with some pages containing more text while others contain more images. Some common elements appear on each page, including a blue left column, a white background, a PC Magazine logo and other small images. The JavaScript cycles through the page loads twice, resulting in a total of 108 web pages being downloaded during this test. When the benchmark is run from a thin client, the thin server would execute the JavaScript that sequentially requests the test pages from the i-Bench server and relay the display information to the thin client. For the web benchmark used in our tests, we modified the original i-Bench benchmark's JavaScript call to introduce delays of several seconds between pages using the JavaScript, sufficient in each case to ensure that the thin client received and displayed each page completely and that there was no temporal overlap in transferring the data belonging to two consecutive pages. We used the packet monitor to record the packet traffic for each page, and then used the timestamps of the first and last packet associated with each page to determine the download time for each page.

We used Netscape Navigator 4.72 as the web client for the web benchmark, as it is available on all the platforms in question. The browser's memory cache and disk cache were enabled but cleared before each test run. In all cases, the Netscape browser window was 1024x768 in size, so the region being updated was the same on each system.

### 2.3.2 Video Benchmark

The video benchmark program processes and displays an MPEG1 video file containing a mix of news and entertainment programming. We measured video performance by monitoring resulting packet traffic at two playback rates, 1 frames/second (fps) and 24 fps. Although no user would want to play video at 1 fps, we took the measurement at that frame rate in order to establish the reference data size transferred from the thin server to the client that corresponds to a "perfect" playback. To measure the normal 24 fps playback performance and video quality, we monitored the packet traffic delivered to the thin client at this playback rate and compared the total data transferred to the reference data size. The video quality can then be quantified by the ratio of data transfer rate at the full frame rate of 24 fps to the transfer rate at the slow-

motion playback rate of 1 fps expressed in percent [37]. The ratio was computed as follows:

$$VQ = \frac{\left[ \frac{\text{DataTransferred}(24 \text{ fps}) / \text{PlaybackTime}(24 \text{ fps})}{\text{IdealFPS}(24 \text{ fps})} \right]}{\left[ \frac{\text{DataTransferred}(1 \text{ fps}) / \text{PlaybackTime}(1 \text{ fps})}{\text{IdealFPS}(1 \text{ fps})} \right]}$$

For the video benchmark, we used two different MPEG1 players. We used Microsoft Windows Media Player version 6.4.09.1109 for the Windows-based thin clients and MpegTV version 1.1 for the Linux/Solaris-based platforms. Both players were used with non-video portions of the interfaces minimized so that the appearance of the playback application was similar across all platforms. In the minimized mode, accessory components like progress bars, frame counters, or clocks were not displayed. The test video clip was 34.75 seconds long and consisted of 834 352x240 pixel frames with an ideal frame rate of 24 fps. The total video file size was 5.11 MB. The thin server executed the video playback program to decode the MPEG1 video then relayed the resulting display to the client.

### 3. Experimental Results

We ran the web and video benchmarks on each of the six thin-client platforms and measured their resulting performance under five network bandwidths. The web benchmark results are shown both in terms of latencies and the respective amounts of data transferred from server to client to illustrate both the overall user-perceived performance and the bandwidth efficiency of the thin-client systems. The data transferred from client to server was not significant in any of our experiments.

Section 3.1 discusses the results obtained for running the thin-client systems with their default configuration options as discussed in Section 2.2. Section 3.2 analyzes the impact of the underlying baseline remote display encodings. Section 3.3 considers the impact of caching and compression mechanisms on thin-client performance.

#### 3.1 Default Configurations

The results of running the web benchmark on each of the thin-client systems with the default settings are shown in Figure 1 through Figure 4. The results of running the video benchmark on each of the thin-client systems are shown in Figure 5 through Figure 8. For comparison purposes, we also show results for using the PC client connected directly through the network emulator to the web and video server to demonstrate the

performance of a traditional “fat” client system for web browsing and streaming video, respectively.

#### 3.1.1 Web Performance

Figure 1 shows the average download latency per page. Usability studies have shown that web pages should take less than one second to download for the user to enjoy an uninterrupted browsing experience [16, 17]. Using this metric, all of the thin-client systems delivered good performance over the 10 Mbps and 100 Mbps LAN bandwidths with average web page latencies well under a second. Using the 100 Mbps bandwidth, X and AIP are the fastest with average web page latencies of less than 300 ms while the other thin-client systems have average latencies of about 500 ms. Figure 1 shows that reducing the bandwidth had the biggest negative impact on X and Sun Ray. In contrast, Citrix ICA, Microsoft RDP, Tarantella AIP, and VNC were able to deliver sub-second average web page latencies over bandwidths as low as 768 Kbps, corresponding to DSL environments. However, none of the thin-client systems were able to deliver sub-second performance at 128 Kbps. Only the PC fat-client achieved sub-second performance across all bandwidths tested. The results indicate that thin-client systems can provide good web browsing performance in broadband or higher bandwidth network environments, but are not yet able to perform well in lower-bandwidth dialup modem and ISDN environments.

The web performance of the systems at various bandwidths can be better understood by examining the average amount of data sent per web page shown in Figure 2. Since the visual quality is constant across all bandwidths as a result of slow-motion benchmarking, the amount of data transferred for each platform is also essentially constant across all bandwidths, except for AIP. For AIP, the different data transfer amounts across various bandwidths is caused by adaptive compression mechanisms which we discuss further in Section 3.3.

At higher bandwidths, there is little correlation between the amount of data transferred and the average web page latency. The best performing thin-client systems at the LAN bandwidths were X and AIP, which sent far more data than the lesser performing ICA and VNC. X sent more data than any other thin-client system except Sun Ray at 100 Mbps, yet it achieved the best performance at this bandwidth. At lower bandwidths, however, there is direct correlation between the amount of data transferred and the average web page latency. ICA sends the least amount of data and has the best performance of all the thin-client systems when using the 128 Kbps network environment. As shown in Figure 2, ICA sends on average about 30 KB of data per page, only twice as

much data for its display updates compared to using HTTP with a PC fat-client.

Figure 3 and Figure 4 show the network bandwidth and client and server CPU utilizations for the web benchmark. The utilization measurements shown do not include the idle time between web pages. Figure 3 shows that the stronger correlation between latency and data transfer efficiency at lower bandwidths is due to the network becoming the main bottleneck. When the average bandwidth utilization exceeds 85 percent, the latency incurred for the thin-client systems generally increases beyond the one-second web page latency threshold. Figure 4 shows the client and server CPU load when using the 100 Mbps network environment. The measurements show that, except for VNC, the clients were not heavily loaded during the web benchmark, indicating that the client CPU was not the primary bottleneck even at high bandwidths. In the case of VNC, the client does not rest much as it is constantly pulling from the server. The CPU utilization for the Sun Ray hardware client is not shown because there were no tools available to measure it. In general, the server CPU was more heavily loaded than the client CPU. AIP, which requires running a web server on the server, had the highest server CPU utilization and appears limited by server speed in a 100 Mbps network environment.

### 3.1.2 Video Performance

Figure 5 shows the resulting video quality on each system for various network bandwidth environments. The video quality was quantified using the VQ formula discussed in Section 2.3.2. Unlike the web benchmark performance, several of the thin-client platforms, ICA, RDP, and VNC, deliver poor video quality even in the 100 Mbps network environment. Only X, AIP, and Sun Ray deliver good video quality at the highest bandwidth. None of the platforms deliver reasonable video quality at lower network bandwidths. Figure 5 shows that X, AIP, and Sun Ray all deliver over 90 percent video quality at 100 Mbps, but that even the best of them degrades to only about 50 percent video quality at 10 Mbps. Sun Ray has a special color space convert display primitive that can be used to improve the video playback performance if the application is written to exploit the feature. The MpegTV application we used, however, was not written to do so. No video benchmark data is shown for Sun Ray at 128 Kbps, because Sun Ray could not play the entire clip without interruption due to the limited bandwidth. The PC fat-client provides good video quality even at 1.5 Mbps, but the video quality rapidly deteriorates at lower bandwidths. For all platforms, the video playback time was relatively constant across all bandwidths, taking about 35 seconds to play the entire video clip.

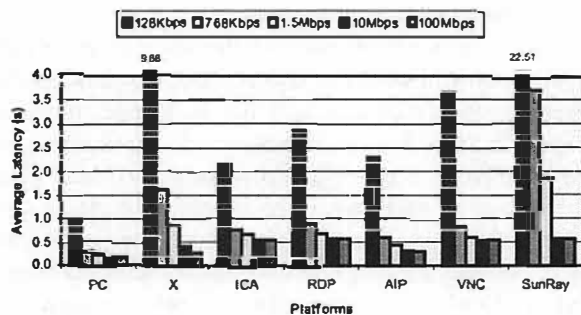


Figure 1: Average latency per page in the web benchmark with default settings at various network bandwidths.

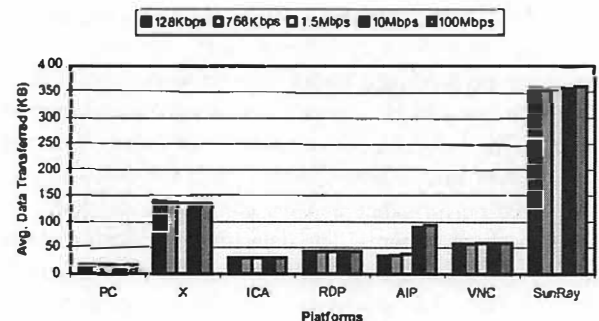


Figure 2: Average data transferred per page in the web benchmark with default settings at various network bandwidths.

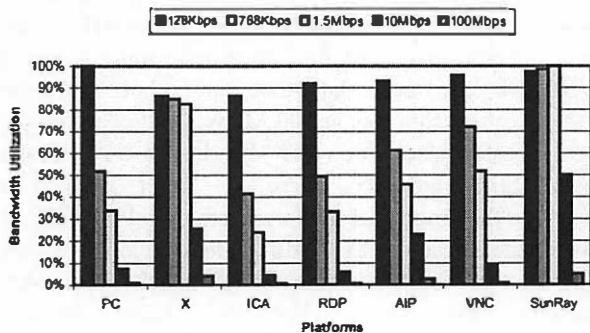


Figure 3: Average bandwidth utilization while downloading pages in the web benchmark with default settings at various network bandwidths.

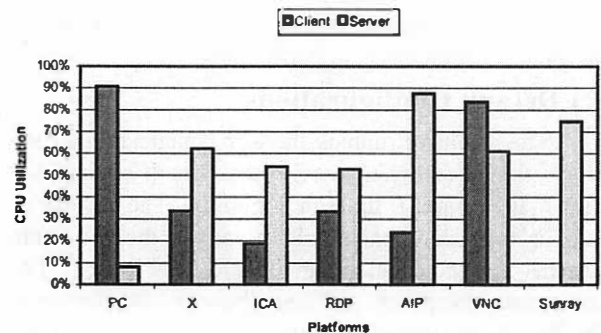
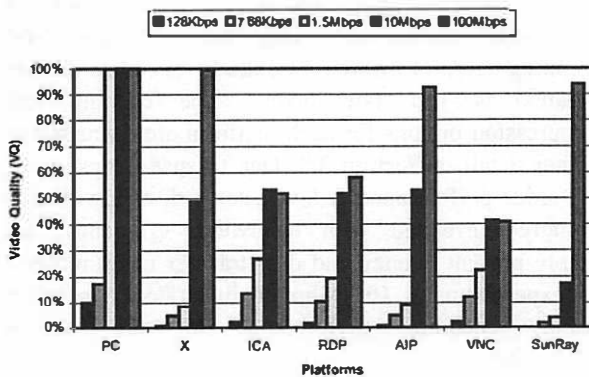


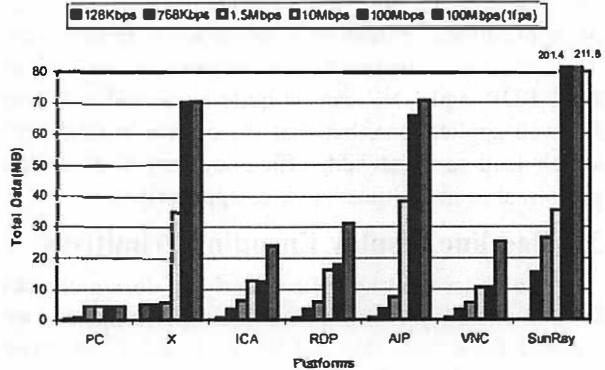
Figure 4: Average client and server CPU utilization while downloading pages in the web benchmark with default settings at 100 Mbps.





**Figure 5:** Video quality in the video benchmark with default settings at various network bandwidths.

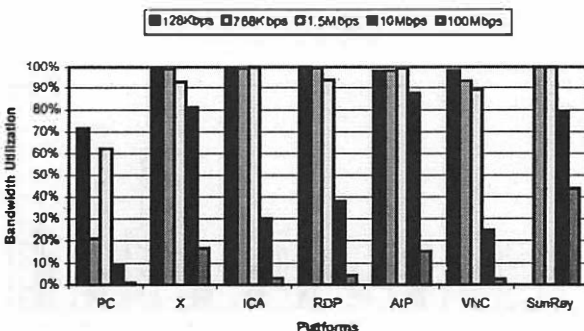
The video performance of the various systems at different bandwidths can be better understood by examining the total data transferred. Figure 6 shows the amount of data transferred by each system at the normal playback rate of 24 fps and at the slow-motion playback rate of 1 fps. The 1 fps data transfer measurements show how efficiently each system encoded the display updates when all of the video frames were fully delivered and displayed on the client. Comparing the 24 fps and 1 fps measurements, we see that all of the systems discard data at lower bandwidths to maintain a constant playback rate, resulting in lower video quality. ICA, RDP, and VNC even discard large amounts of data at 100 Mbps. Figure 6 also shows that the thin-client systems that performed the best on the video benchmark were also the least data efficient at encoding the display. AIP and X transferred roughly 70 MB to play back the video clip in 8-bit color and Sun Ray transferred roughly three times that amount to display in 24-bit color. These data transfer rates are comparable to sending raw pixels over the network for each 352x240 pixel frame and more than ten times the transfer rate of MPEG streaming the 5.11 MB clip.



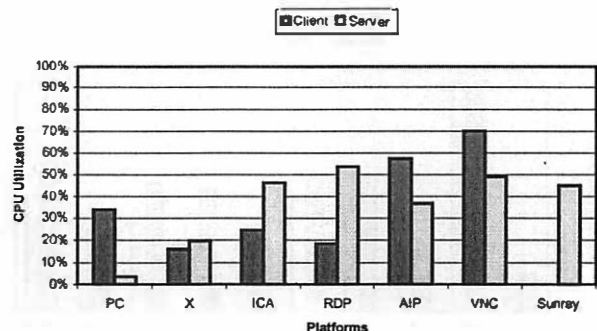
**Figure 6:** Total data transferred in full-motion (24 fps) and slow-motion (1 fps) playback in the video benchmark with default settings at various network bandwidths.

Comparing Figure 7 and Figure 3 shows that the average bandwidth consumption of the thin-client systems when running the video benchmark was much higher than when running the web benchmark. None of the platforms was bandwidth limited at 100 Mbps, even though half of the systems (ICA, RDP, and VNC) delivered poor video quality at that bandwidth. However, all of the three systems (X, AIP, and Sun Ray) that delivered good video quality at 100 Mbps consumed well over 10 Mbps of network bandwidth. As a result, bandwidth limitations were the primary bottleneck for these three systems at lower network bandwidths. For the other systems that failed to perform well even at 100 Mbps, Figure 8 indicates that none of the client or server systems had high CPU load except for VNC. We note that while none of the client and server average utilization measurements reached 100 percent, there was high variability in the system loads with frequent peaks at 100 percent for VNC on the client-side, suggesting that VNC video performance appears to be limited by the client's CPU speed.

Our measurements of thin-client performance on the web and video benchmarks indicate that AIP, X,



**Figure 7:** Average bandwidth utilization during video playback in the video benchmark with default settings at various network bandwidths.



**Figure 8:** Average client and server CPU utilization during video playback in the video benchmark with default settings at 100Mbps.

and Sun Ray are more able to support a broader range of applications, particularly multimedia applications. The results also suggest that thin-client systems such as ICA, RDP, and VNC can be quite bandwidth efficient for web applications, but that these same mechanisms which lead to bandwidth efficiency may degrade the performance in multimedia video applications.

### 3.2 Baseline Display Encoding Primitives

To understand how the underlying design choices in thin-client systems impact their performance, we isolated the effects that can be attributed to the basic display encoding primitives used. Four types of display encoding primitives are high-level graphics, low-level graphics, 2D draw primitives, and raw pixels. Higher-level display encodings are generally considered to be more bandwidth efficient, but may require more computational complexity on the client and may be less platform-independent. For instance, graphics primitives such as fonts require the thin-client system to separate fonts from images while using pixel primitives enable the system to view all updates as just regions of pixels without any semantic knowledge of the display content. X takes a high-level graphics encoding approach and supports a rich set of graphics primitives in its protocol. ICA, RDP, and AIP are based on lower-level graphics primitives that include support for fonts, icons, drawing commands as well as images. Sun Ray and VNC employ 2D draw primitives such as fills for filling a screen region with a single color or a two-color bitmap for common text-based windows. VNC can also be configured to use raw pixel encoding only, but none of the systems we considered used raw pixels by default.

To examine the basic display encoding performance, we disabled all configurable caching and compression mechanisms and ran the benchmarks. For AIP, there was no option to disable caching. For VNC, the display compression could not be disabled because it is built into the default hextile display encoding used. For X and Sun Ray, the baseline and default configurations were the same as there were no caching

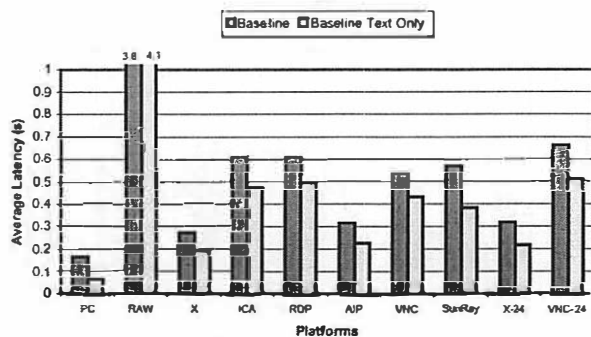
and compression options. For comparison purposes, we also show measurements using the VNC raw pixel encoding (RAW), which essentially encodes display updates as just raw pixels. The caching and compression options for each platform are discussed in further detail in Section 3.3. Due to space constraints, and since performance at lower network bandwidths is strongly correlated with bandwidth efficiency, we simply present latency and data transfer measurements for experiments at 100 Mbps to illustrate the baseline display encoding performance for the various approaches.

#### 3.2.1 Web Performance

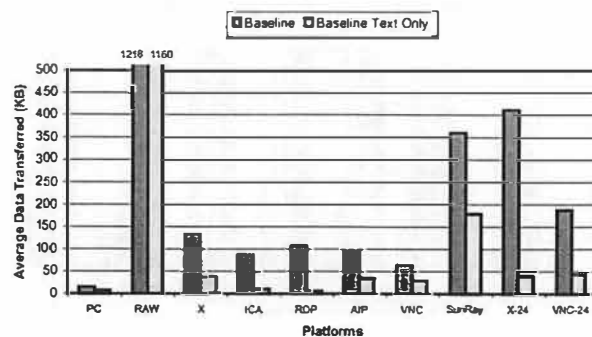
Figure 9 and Figure 10 show the latency and data transfer measurements for the baseline performance of the thin-client systems running the web benchmark. In particular, we show results for running two versions of the web benchmark: one with all of the images displayed normally, and one with just text in which all of the images were removed and replaced with blank spaces of equal size. We employed both versions to compare how different thin-client mechanisms perform on graphics versus text-oriented media.

We first discuss the baseline measurements with the standard benchmark content (both images and text). Figure 9 shows that the average web page download latencies are not much different than those with the default thin-client configurations discussed in Section 3.1.1. We note that all of the systems fare much better than RAW, which results in unacceptable average web page latencies of over 4 seconds. ICA and RDP exhibit somewhat higher latencies using just the baseline display encoding primitives as opposed to the default configurations. X and AIP still deliver the lowest average web page download latencies.

The more interesting measurements are in Figure 10, which shows the average data transferred per web page at the baseline settings. Comparing with RAW, the results show that all of the other display encodings used are substantially more bandwidth efficient than sending



**Figure 9:** Average latency per page in the web benchmark with baseline settings at 100 Mbps.



**Figure 10:** Average data transferred per page in the web benchmark with baseline settings at 100 Mbps.

raw pixels, in some cases by more than an order of magnitude. ICA, RDP, and AIP all send about the same amount of data, which is consistent with the fact that they all employ low-level graphics display encoding primitives. X, which employs the higher-level graphics primitives, surprisingly sends the most data among all the 8-bit color thin-client systems. Although both VNC and Sun Ray use 2D draw primitives, the amount of data sent in each case is quite different. While the VNC display encoding appears the most data efficient, it includes built-in compression so comparing its efficiency with the other systems without compression is not a fair comparison. On the other hand, Sun Ray uses 24-bit color, so comparing its efficiency with other 8-bit systems is not entirely fair either.

To account for the impact of different color depths on display encoding efficiency, we also measured the performance of X and VNC using 24-bit color, as these were the only platforms we used that could operate using either 8-bit or 24-bit color depth. As shown in Figure 10, both X and VNC send roughly three times as much data using 24-bit color as opposed to using 8-bit color. This suggests that to fairly compare Sun Ray with the other 8-bit color results, we should normalize the amount of data transferred by the pixel color depth, which would effectively reduce the amount of data Sun Ray transferred by a factor of three. The normalized Sun Ray data transfer measurements would then be better than X and only about 20 percent worse than ICA. Surprisingly, the use of simple 2D draw primitives results in data transfer requirements better than the high-level graphics X approach and not much different from the low-level graphics approach used by ICA, RDP, and AIP. Furthermore, Figure 9 shows that Sun Ray performs somewhat better than the 8-bit color ICA and RDP platforms despite providing a higher quality 24-bit color display.

Figure 9 and Figure 10 also show the latency and data transfer measurements for the performance of the thin-client systems running the text-only version of the

web benchmark. These results suggest that the higher-level display encodings are more optimized to reduce the data transfer requirements of text content as opposed to image content. Figure 10 shows that the higher-level encodings used by ICA, RDP, AIP, and X were much more bandwidth efficient for text than the lower-level encodings used by Sun Ray and VNC. In particular, RDP reduced the amount of data sent for text to less than five percent of that for both images and text. Despite the large bandwidth savings for text content, the higher-level encoding systems do not provide the same degree of reduction in latency, as shown in Figure 9. Instead, Sun Ray demonstrates the largest percentage reduction in web page download latency despite having the smallest percentage reduction in the amount of data transferred when comparing image and text content to text-only content. This again demonstrates that at a high enough bandwidth, the encoding overhead rather than the amount of data generated is the primary factor in determining the performance.

### 3.2.2 Video Performance

Figure 11 and Figure 12 show the video quality and data transfer measurements for the baseline performance of the thin-client systems. The video quality results shown in Figure 11 for the baseline display encoding configuration are quite similar to the results for the default configuration discussed in Section 3.1.2. All of the systems performed much better than RAW, which yielded poor video quality of less than 15 percent. X, AIP, and Sun Ray still deliver good video quality while ICA, RDP, and VNC deliver noticeably worse video quality. Although the video quality for ICA and RDP are similar to their respective performance with the default configurations, Figure 12 shows that they send roughly twice as much data when just using the basic display encoding.

To account for the impact of different color depths on display encoding efficiency, we again measured the performance of X and VNC using 24-bit color as well.

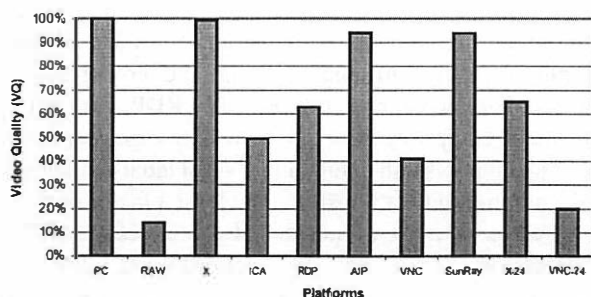


Figure 11: Video quality in the video benchmark with baseline settings at 100 Mbps.

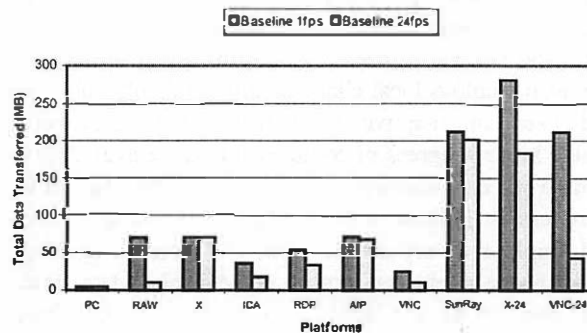


Figure 12: Total data transferred in full-motion (24 fps) and slow-motion (1 fps) playback in the video benchmark with baseline settings at 100 Mbps.

As expected, both platforms send substantially more data using 24-bit color versus using 8-bit color. In addition, Figure 11 shows that when using 24-bit color, the video quality of VNC remains poor and the video quality of X decreases down to about 65 percent. When comparing among the 24-bit color platforms, Sun Ray clearly delivers the best video quality.

An important lesson derived from the default and baseline video benchmark results is that the timing of display update can be just as important as how a display update is encoded. X, AIP, and Sun Ray employing an eager server-push display update model excelled in the video benchmark at 100 Mbps. AIP also uses a lazy model to adapt to lower bandwidths. When a rendering command is generated by the application, these thin-client systems immediately convert that command to the underlying display encoding primitives and send the display update to the client. The eager updates enable the server to keep up with the video application's rendering commands and allow the server to take advantage of any semantic information that can be used from the rendering command. In contrast, ICA, RDP, and VNC employ a lazy display update model, in which multiple rendering commands are first buffered and then later merged before lazily sending the merged display updates to the client. For ICA and RDP, the updates are lazily sent at a server-defined rate. The problem is that the updates are not sent frequently enough for real-time video display, resulting in multiple video frames being merged and overwritten at the server and never displayed at the client. For VNC, the updates are lazily sent when the client requests them. Since the client running VNC is already heavily loaded, the client becomes a bottleneck in requesting the display updates, resulting in lost video frames that are merged and overwritten at the server before the client is able to generate the next display request.

### 3.3 Caching and Compression

Four of the six thin-client platforms tested employ some form of configurable caching or compression to improve system performance. ICA and RDP both employ run-length encoding compression and cache fonts and bitmaps in memory and on disk at the client. AIP also employs local client caching of display objects and uses an adaptive mechanism to progressively enable higher-degrees of compression as the availability of network bandwidth becomes limited. VNC has RLE compression built-in with its display encoding format and employs a very simple form of on-screen caching whereby the client can simply copy display data from one portion of the screen to another rather than requesting it from the server if the display data is already displayed on another portion of the framebuffer.

To examine the performance impact of caching and compression techniques, we measured the performance of the thin-client systems on the web and video benchmarks with various caching and compression configuration settings. We show results for ICA, RDP, AIP, and VNC. In Section 3.3.1 and 3.3.2, we compare four configurations: (1) the baseline results from Section 3.2 with all caching and compression options disabled, (2) all compression only options enabled, (3) all caching only options enabled, and (4) all caching and compression options enabled. In particular, for ICA and RDP which support both memory and disk caching, we enabled or disabled both caches together. In Section 3.3.3, we explore the disk and memory caching options of ICA separately in further detail. For AIP, there was no option to disable caching as mentioned in Section 3.2, so the AIP cache only and baseline and cases are the same and there was no compression only configuration tested. For VNC, the compression cannot be separately configured as it is part of the default hextile encoding used, so the VNC baseline and compression only cases are the same and there was no cache only configuration tested.

Figure 13 through Figure 16 show the latency and data transfer measurements for running the web benchmark relative to the baseline performance of each system as reported in Section 3.2.1. We again show results for running the normal web benchmark with both images and text and the text-only version of the web benchmark. Figure 17 and Figure 18 show the video quality and slow-motion 1 fps data transfer measurements for running the video benchmark relative to the baseline performance of each system as reported in Section 3.2.2.

#### 3.3.1 Web Performance

Figure 13 shows that using 100 Mbps bandwidth, there is no significant performance benefit due to caching and compression options in most of the thin-client systems. The most notable difference occurs for ICA with caching enabled. Surprisingly, enabling ICA's cache increases the average web page latency by almost 40 percent over the baseline performance.

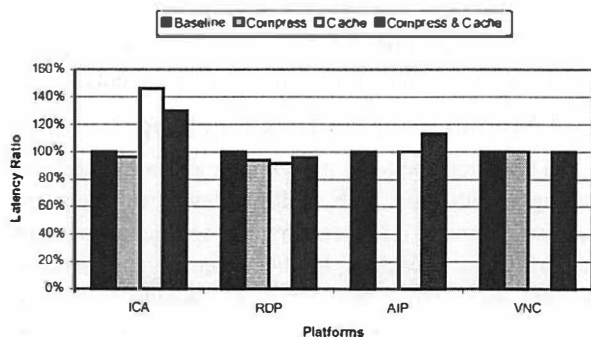
Figure 14 shows that there was a substantial difference in the amount of data transferred for almost all platforms for different caching and compression options. For all three platforms, ICA, RDP, and AIP, for which compression could be enabled or disabled, enabling compression resulted in a substantial reduction in the amount of data transferred, at least a factor of two in all cases. It must be noted that the effect of AIP's compression could not be isolated and directly compared with those of RDP and ICA, because its cache could not be disabled. But AIP seems to have a

large reduction in data transfer when its compression is engaged, which is most likely due to its use of both RLE and LZW compression as opposed to using only RLE compression for ICA and RDP. AIP, however, was adversely affected by the added processing overhead of using cache and compression at 100 Mbps. When compression was enabled, the latency increased by 13%. At higher bandwidths, where the network is not the bottleneck, it may be advantageous to reduce the processing overhead by holding back on compression even if it results in a larger amount of data. Since performance at lower bandwidths is directly related to the amount of data transferred, compression is beneficial for improving performance at lower bandwidths.

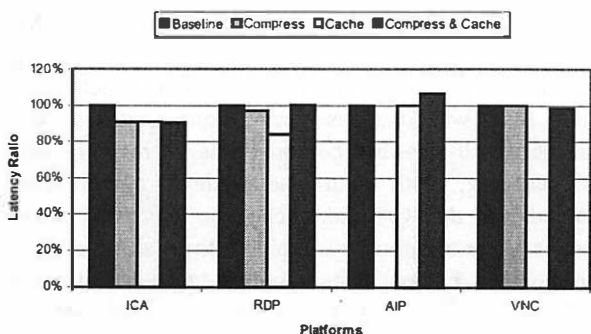
Caching is also not always beneficial. Among the systems that provided the option to enable or disable caching, Figure 14 shows that enabling caching results in the largest reduction in data transferred for ICA. ICA shows almost a factor of three reduction in data transfer for just using caching, and yet results in a significant increase in the average web page latency. In other words, the overhead of ICA caching outweighs its benefits in high bandwidth network environments. On

the other hand, using caching with RDP and VNC resulted in very little difference in either latency or data transferred versus not using caching. For VNC, the on-screen cache contains only the current display data which does not provide sufficient history to be beneficial in reducing the amount of data that the server needs to send. However, the ineffectiveness of the cache for RDP is more surprising as its caching architecture is similar to ICA on the surface. Our results indicate that RDP's caching mechanism may not be operating correctly at best or poorly designed at worst. Figure 14 and Figure 16 show that there was no reduction in data size due to RDP's disk cache.

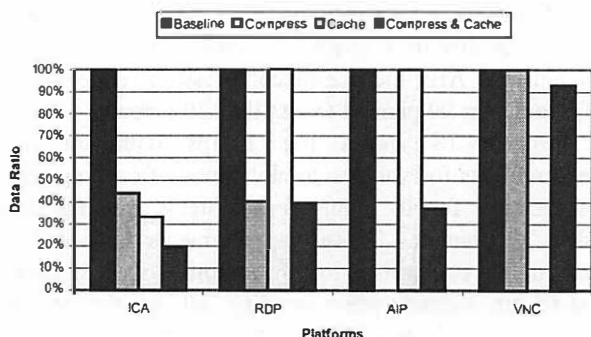
Figure 15 and Figure 16 show the latency and data transfer measurements for various combinations of caching and compression for the thin-client systems running the text-only web benchmark. The results for running the text-only benchmark were generally similar to those for the normal web benchmark with both text and images. These results suggest that the caching and compression mechanisms have similar advantages and disadvantages for both the image and text content of the web benchmark. The one exception was for using caching with ICA. With the text-only content, the



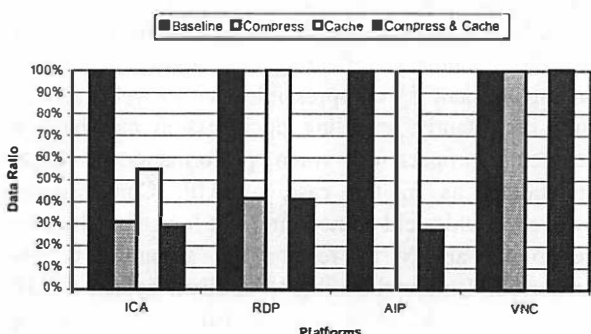
**Figure 13:** Latency (expressed as percentage relative to baseline) in the web benchmark at 100 Mbps with various cache and compression settings.



**Figure 15:** Latency (expressed as percentage relative to baseline) in the web benchmark at 100 Mbps with various cache and compression settings and with only text content.

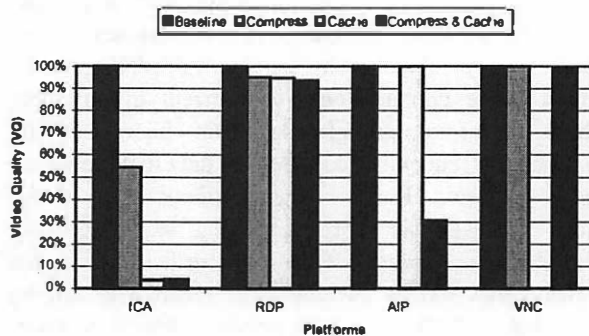


**Figure 14:** Data transferred (expressed as percentage relative to baseline) in the web benchmark at 100 Mbps with various cache and compression settings.



**Figure 16:** Data transferred (expressed as percentage relative to baseline) in the web benchmark at 100 Mbps with various cache and compression settings and with only text content.





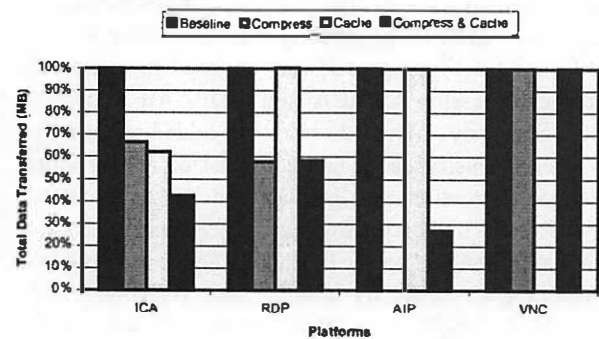
**Figure 17:** Video quality (expressed as percentage relative to baseline) in the video benchmark at 100 Mbps with various cache and compression settings.

performance did not degrade when ICA's caching was engaged as we saw with both text and images.

### 3.3.2 Video Performance

Figure 17 shows the video quality measurements for various combinations of caching and compression for the thin-client systems running the video benchmark at 100 Mbps. For RDP and VNC, there was little difference in the video quality for the various options. For ICA, the biggest difference again appeared with the use of caching, which resulted in a substantial decrease in video quality from roughly 50 percent to less than 5 percent. For AIP, the use of compression reduced the VQ from over 90 percent to less than 30 percent.

Figure 18 shows the 1 fps data transfer measurements for various combinations of caching and compression for the thin-client systems running the video benchmark. These measurements provide a quantitative comparison of the amount of data each system transferred when sending all of the video content to the client without discarding data. Just as for the web benchmark, for all three platforms, ICA, RDP, and AIP, for which compression could be enabled or disabled, enabling compression resulted in a substantial reduction in the amount of data. The data reduction was generally not as large for the video benchmark as for the web benchmark, reflecting the fact that the video content was not as compressible as the web content. More importantly, enabling compression can have a detrimental impact on video performance at LAN bandwidths, as in the case of AIP. Compression, however, could yield some benefit at lower bandwidths due to its ability to reduce the amount of data transferred. Unlike the other thin-client systems, AIP employs an adaptive mechanism for enabling compression that turns compression off at high bandwidths and on at low bandwidths. Our results suggest that an adaptive mechanism for enabling compression at lower bandwidths is useful in trading



**Figure 18:** Total data transferred (expressed as percentage relative to baseline) in slow-motion (1 fps) playback in the video benchmark at 100 Mbps with various cache and compression settings.

off compression overhead versus bandwidth savings at different bandwidths.

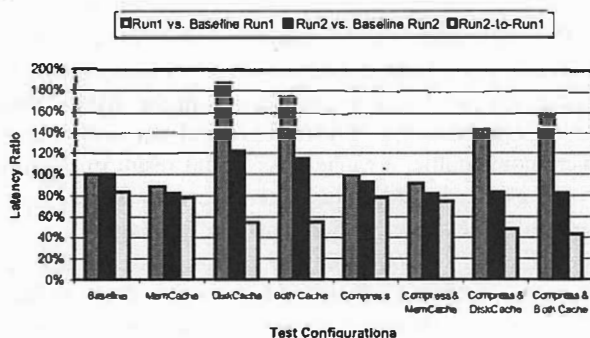
As in the case of the web benchmark, caching did not consistently reduce the amount of data transferred for the video benchmark. Among the systems that provided the option to enable or disable caching, Figure 18 shows that enabling caching reduced the amount of data transferred for ICA, but had no impact on the amount of data transferred for RDP or VNC. Just as with the web benchmark, the video benchmark results indicate that the overhead of ICA caching outweighs its benefits in high bandwidth network environments.

### 3.3.3 Memory versus Disk Caching

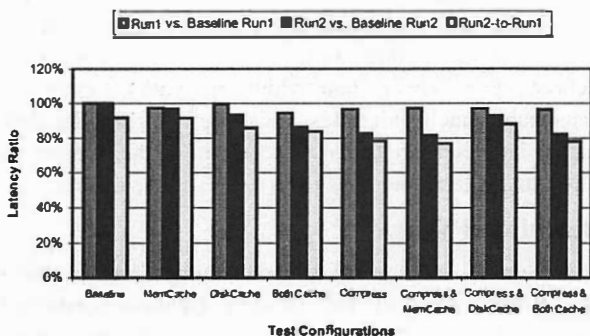
Thin-client systems may implement a hierarchical caching architecture with multiple levels of cache. In ICA, two forms of client caching are applied to improve the performance: caching in client memory and caching in client disk. These two forms of caching may have very different characteristics. Memory caching can provide much faster access times to smaller caches while disk caching can provide larger amounts of local cache with relatively slower access times. ICA provides both memory and disk caching as well as the ability to enable and disable each cache independently. We investigated the impact of memory and disk caching techniques by running the web and video benchmarks using ICA with various cache configurations. We considered all possible combinations of memory and disk caching, both with and without compression enabled. For the ICA disk cache, the maximum cache space and the minimum cacheable bitmap size are user-configurable. For our tests, the disk cache size was set to 39 MB, and the minimum cacheable bitmap size to 8KB. The memory cache size was 8 MB. These disk and memory cache settings were default in the ICA client.

Figure 19 through Figure 22 show the performance of ICA with various cache and compression combinations available for ICA. As discussed in Section 2.3.1, the web benchmark cycles through 54 web pages twice. We call the first iteration Run1 and the second Run 2. In order to highlight the effects of caching and compression, we present the performance relative to the baseline configuration as well as the performance ratio of Run 2 to Run 1. If enough elements are cached while displaying the content from Run 1, we would expect the Run 2 to produce less data transferred from server to client and potentially yield a better performance. Also, if some elements are displayed repeatedly within the 54-page iteration, then we would expect the transferred data amount to decrease in Run 1 as well as Run2.

While there is no tool available to us to directly measure the cache hit/miss rate reliably for ICA, it would be reasonable to assume the ratio of data transferred from the server to the client with cache turned on to that with cache off provides a rough measure of the cache miss rate. As shown in Figure 20,



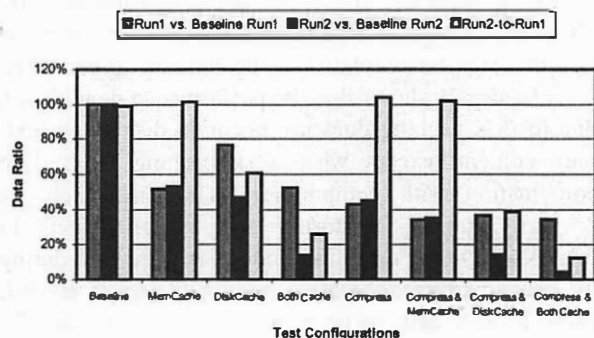
**Figure 19:** The latency in Run 1 and Run 2 of the web benchmark at 100 Mbps with various cache and compression settings in ICA. The Run 1 and Run 2 latency are expressed as percentage relative to baseline as well as relative to one another.



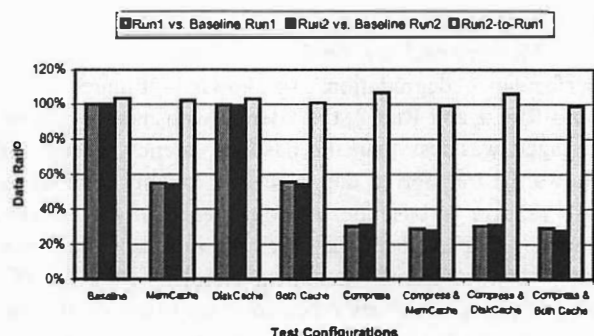
**Figure 21:** The latency in Run 1 and Run 2 of the web benchmark at 100 Mbps with various cache and compression settings in ICA and with only text content. The Run 1 and Run 2 latency are expressed as percentage relative to baseline as well as relative to one another.

Run 1 of the benchmark run at 100 Mbps with disk cache on produced 77% of the data generated by Run 1 with the baseline configuration. That is, the client was forced to fetch 77% of the total display data from the server even with the disk cache on, presumably because the data wasn't found in the local cache. In Run 2, however, the data ratio drops to 48%. As expected, more data was found in the local cache in Run 2. Inferring from Figure 2, the first iteration of 54 pages would yield only 1.6 MB of data, which would fit well within the cache. However, not all of the elements were cached even though the 39 MB disk cache had enough capacity to store all objects encountered in Run 1. In particular, the bitmap objects smaller than 8 KB were not cacheable per the disk cache setting we used.

Comparing the relative data size and latency between Run 1 and Run 2, it is evident that the memory cache serves to handle small elements, while the disk cache is used for caching large bitmaps. Figure 19 shows that there is less significant improvement in latency in Run 2 compared to Run 1 with memory cache engaged. Figure 20 shows that there is almost no



**Figure 20:** The data size in Run 1 and Run 2 of the web benchmark at 100 Mbps with various cache and compression settings in ICA. The Run 1 and Run 2 data sizes are expressed as percentage relative to baseline as well as relative to one another.



**Figure 22:** The data size in Run 1 and Run 2 of the web benchmark at 100 Mbps with various cache and compression settings in ICA and with only text content. The Run 1 and Run 2 data sizes are expressed as percentage relative to baseline as well as relative to one another.

difference in data size between Run 1 and Run 2. While small graphical elements appear repeatedly throughout each cycle of 54 web pages, the large bitmaps seen in Run 1 only reappear when the same web page reappears in Run 2. If the memory cache cached larger objects, then we would expect to see a significant change in Run 2 compared to Run 1. With disk caching, however, we do observe such a change. The difference in the types of objects cached caused the two methods of cache to yield very different performance characteristics in our tests.

A notable finding was that, at 100 Mbps, ICA performed worse whenever the disk cache was engaged even though the cache significantly reduced the amount of transferred data. As shown in the web benchmark results in Figure 19, the increase in latency with disk caching, relative to the baseline setting, was almost by a factor of two in Run 1. In Run 2, there was a slight improvement in performance with the disk cache engaged, but when accounting for both Run 1 and Run 2, there was 44% higher latency overall. These data suggest there is a heavy cache-miss penalty associated with ICA's disk caching. At a high bandwidth like 100 Mbps, the amount of time required to look up the cache becomes significant relative to the network access time.

Figure 21 shows that the performance degradation due to disk caching does not occur in displaying text-only content, except when disk caching is used in combination with compression. The disk cache is primarily utilized for storing large bitmap objects. In the text-only test, no bitmap image is displayed during the benchmark run; therefore, we would expect the disk cache to have little to no effect. As seen in Figure 22, the disk cache does not contribute to any decrease in data size. We note that, in general, the data size increases slightly in Run 2 of the text-only tests compared to Run 1, because Netscape for Windows, with its own cache engaged, behaves slightly differently in Run 2 compared to Run 1 in terms of the way the page is drawn.

Memory caching, on the other hand, introduced no performance degradation. As shown in Figure 19, in both Run 1 and Run 2, the latency with memory cache engaged was less than the baseline latency. Figure 20 shows the transferred data size was roughly reduced to half relative to baseline. Although each of the 54 web pages is displayed for the first time in Run 1, there are fonts, text, and small graphical elements (like the PC Magazine logo) that are repeated many times. With disk caching, any benefit in caching the repeated graphical elements was overwhelmed by the penalty in looking up the cache on the hard drive. The cache-miss penalty associated with the memory cache is much less severe.

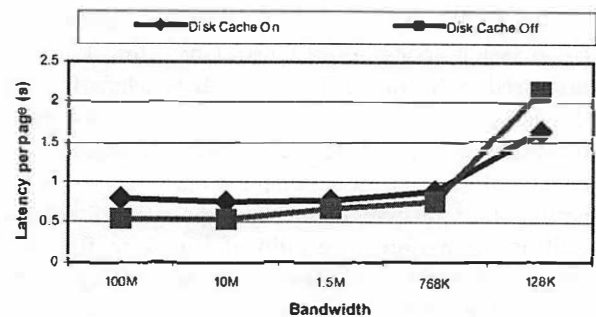


Figure 23: Average latency per page in the web benchmark for ICA with disk cache on and off.

The lower thin-client performance with disk caching is due to the relative speed of the network compared with the disk and the penalty associated with cache misses. In a 100 Mbps LAN environment, the network speed is almost comparable in speed and bandwidth to the sustained performance of the local disk of our client machine. Consequently, obtaining display data from the disk cache is not necessarily faster than obtaining the data from the server across the 100 Mbps network. In addition, with disk caching enabled, each disk cache miss requires the client to access the local disk as well as obtain the display data across the network. If local disk and network speeds were comparable, a cache miss would result in roughly twice as much latency as when the data were simply sent from the server without any disk caching.

Figure 23 compares the performance of ICA at various network bandwidths with the default configuration settings versus the same settings except with the disk cache enabled. The results show that while disk caching adversely affects ICA performance at higher network bandwidths, it improves ICA performance at bandwidths below 768 Kbps. At low enough network bandwidths, the disk access time becomes insignificant relative to the network access time such that it is much faster to fetch data from the client disk cache than going across the network to the server. For lower bandwidth networks, assuming reasonable cache hit rates, the benefit of smaller disk cache latencies on cache hits outweigh the penalty of extra disk cache latencies incurred on cache misses.

#### 4. Related Work

Several studies have been conducted to evaluate thin-client computing architectures. Danskin conducted an early study of the X protocol [7] by gathering traces of X requests. Citrix and Microsoft have conducted internal performance testing of their products. Microsoft has examined thin-client scalability issues in Terminal Services performance for the purposes of

capacity planning [15]. Schmidt, Lam, and Northcutt examined the performance of the Sun Ray platform in comparison to the X protocol [26]. Wong and Seltzer have studied the performance of Windows NT Terminal Server and LBX [34, 35]. Tolly Research has conducted similar studies for Citrix MetaFrame [31]. Howard has measured the performance of various hardware thin clients using the i-Bench benchmark suite [9], but his results suffer from methodology problems due to only measuring server-side application performance instead of user-perceived client-side performance. We have also conducted earlier studies of thin-client performance [18, 19, 36, 37], including previously developing the slow-motion benchmarking [37] used in this paper. Some of these studies have examined selected thin-client systems in detail via internal instrumentation. However, no study considered the performance of remote display mechanisms across the broad range of systems, system configurations, and network bandwidths discussed here. We have also further considered the performance of thin-client systems in wide-area network environments [12].

In addition to the thin-client systems discussed in this paper, a number of other systems for remote display have been developed. These include extensions to the systems considered such as low-bandwidth X (LBX) [1] and Kaplinsk's recent VNC tight encoding [11] as well as remote access solutions such as Laplink [13] and PC Anywhere [20]. Because of space constraints and previous work [18, 19] showing that LBX, Laplink, and PC Anywhere perform very poorly, we did not include them in this study. While thin-client systems have primarily been employed in LAN environments, a growing number of ASPs are employing thin-client technology to host desktop computing sessions that are remotely delivered over WAN environments. Examples include services from Charon Systems [3], Runaware [23], and Expertcity [8].

## 5. Conclusions and Future Work

Our results show that thin-client systems can provide good performance for web and multimedia applications in LAN environments. Unlike traditional PC software environments, our results show that different thin-client system designs exhibit widely varying performance that can differ by orders of magnitude in some cases. Through our experiments, we have analyzed various design choices underlying current thin-client systems. Specifically, our measurements show three important conclusions regarding thin-client system design.

First, higher-level graphics display primitives are not always more bandwidth efficient than lower-level display encoding primitives. X, which uses high-level

graphics encoding consumed the most bandwidth in rendering the display at 8-bit color. Furthermore, higher-level primitives are often more optimized for text-oriented content, which will likely become a smaller and smaller percentage of display content as multimedia applications become increasingly popular.

Second, the timing in sending display updates from the server to the client can be as important as how display updates are encoded. Our results indicate that an eager server-push model as used in X and Sun Ray provides better overall performance than lazy update models like ICA, RDP, and VNC, especially for multimedia video applications. While lazy update models may lead to some bandwidth savings by discarding or merging display updates, our results show that these techniques for optimizing bandwidth efficiency degrade the performance of multimedia applications even in high bandwidth environments.

Third, display caching and compression are techniques which should be used with care as they can help or hurt thin-client performance. At higher bandwidths, ICA displayed significant performance degradation when caching was engaged, and AIP slowed down when its compression was forced on. Our results with current thin-client systems suggest that existing compression techniques provide a greater performance benefit than current caching mechanisms. Furthermore, adaptive use of these mechanisms based on the availability of network bandwidth as shown by AIP produces a good balance between the computational overhead of these encoding mechanisms and the potential bandwidth savings that they provide. In general, cutting down the processing time is desirable when there is enough network bandwidth, while reducing the amount of transferred data is beneficial at lower network speeds.

Our results quantify the effectiveness of a number of thin-client design and implementation choices across a broad range of thin-client platforms and network environments. In doing so, we provide the first comparative analysis of the performance of these systems. These measurements provide a basis for future research in developing more effective thin-client systems.

## 6. Acknowledgments

This work was supported in part by an NSF CAREER Award, NSF grant EIA-0071954, and National Semiconductor. We thank Naomi Novik for developing the scripts for processing the raw benchmark data and Brian Schmidt for helpful comments on earlier drafts of this paper.

## 7. References

1. "Broadway / X Web FAQ", <http://www.broadwayinfo.com/bwfaq.htm>.
2. M. Chapman, <http://www.rdesktop.org>.
3. Charon Systems, <http://www.eharon.com>.
4. B. O. Christiansen, K. E. Schausser, and M. Munke, "A Novel Codec for Thin Client Computing", *Data Compression Conference (DCC)*, Snowbird, UT, Mar. 2000.
5. "Citrix ICA Technology Brief", Technical White Paper, Boca Research, Boca Raton, FL, 1999.
6. B. C. Cumberland, G. Carius, and A. Muir, *Microsoft Windows NT Server 4.0, Terminal Server Edition: Technical Reference*, Microsoft Press, Redmond, WA, Aug. 1999.
7. J. Danskin and P. Hanrahan, "Profiling the X Protocol", *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, TN, May 1994.
8. "DesktopStreaming Technology and Security", Expertcity White Paper, Expertcity.com, Santa Barbara, CA, 2000.
9. B. Howard, "Thin Is Back", *PC Magazine*, 19(7), Ziff-Davis Media, New York, NY, July 2000.
10. i-Bench version 1.5, Ziff-Davis, Inc., <http://www.ctestinglabs.com/benchmarks/i-bench/i-bench.asp>.
11. C. Kaplinsk, "Tight encoding", <http://www.tightvnc.com/comparc.html>.
12. A. Lai and J. Nieh, "Limits of Wide-Area Thin-Client Computing", *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Marina del Rey, CA, June 2002.
13. "LapLink 2000 User's Guide", LapLink.com, Inc., Bothell, WA, 1999.
14. T. W. Mathers and S. P. Genoway, *Windows NT Thin Client Solutions: Implementing Terminal Server and Citrix MetaFrame*, Macmillan Technical Publishing, Indianapolis, IN, Nov. 1998.
15. "Windows 2000 Terminal Services Capacity Planning", Technical White Paper, Microsoft Corporation, Redmond, WA, 2000.
16. J. Nielsen, *Multimedia and Hypertext: The Internet and Beyond*, Morgan Kaufmann, San Francisco, CA, Jan. 1995.
17. J. Nielsen, *Designing Web Usability*, New Riders Publishing, Indianapolis, IN, 2000.
18. J. Nieh and S. J. Yang, "Measuring the Multimedia Performance of Server-Based Computing", *Proceedings of the 10<sup>th</sup> International Workshop on Network and Operating System Support for Digital Audio and Video*, Chapel Hill, NC, June 2000.
19. J. Nieh, S. J. Yang, and N. Novik, "A Comparison of Thin-Client Computing Architectures", Technical Report CUCS-022-00, Department of Computer Science, Columbia University, November 2000.
20. PC Anywhere, <http://www.symantec.com/pcanywhere>.
21. Personable.com, <http://www.personable.com>.
22. T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper, "Virtual Network Computing", *IEEE Internet Computing*, 2(1), Jan/Feb 1998.
23. Runaware.com, <http://www.runaware.com>.
24. "Tarantella Web-Enabling Software: The Adaptive Internet Protocol", SCO Technical White Paper, Santa Cruz Operation, Dec. 1998.
25. R. W. Scheifler and J. Gettys, "The X Window System", *ACM Transactions on Graphics*, 5(2), Apr. 1986.
26. B. K. Schmidt, M. S. Lam, and J. D. Northcutt, "The Interactive Performance of SLIM: A Stateless, Thin-Client Architecture", *Proceedings of the 17<sup>th</sup> ACM Symposium on Operating Systems Principles*, Kiawah Island Resort, SC, Dec. 1999.
27. A. Shaw, K. R. Burgess, J. M. Pullan, and P. C. Cartwright, "Method of Displaying an Application on a Variety of Client Devices in a Client/Server Network", US Patent US6104392, Aug. 2000.
28. B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 2<sup>nd</sup> edition, Addison-Wesley, Reading, MA, 1992.
29. "The Cloud", Shunra Software, <http://www.shunra.com>.
30. "Sun Ray I Enterprise Appliance", Sun Microsystems, <http://www.sun.com/products/sunrayI>.
31. Tolly Research, "Thin-Client Networking: Bandwidth Consumption Using Citrix ICA", *IT clarity*, Feb. 2000.
32. Virtual Network Computing, <http://www.uk.research.att.com/vnc>.
33. WildPackets, Inc., Etherpcek 4, <http://www.wildpackets.com>.
34. A. Y. Wong and M. Seltzer, "Evaluating Windows NT Terminal Server Performance", *Proceedings of the 3<sup>rd</sup> USENIX Windows NT Symposium*, Seattle, WA, July 1999, pp. 145-154.
35. A. Y. Wong and M. Seltzer, "Operating System Support for Multi-User, Remote, Graphical Interaction", *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000, pp. 183-196.
36. S. J. Yang and J. Nieh, "Thin Is In", *PC Magazine*, 19(13), Ziff-Davis Media, New York, NY, July 2000.
37. S. J. Yang, J. Nieh, and N. Novik, "Measuring Thin-Client Performance Using Slow-Motion Benchmarking", *Proceedings of the USENIX 2001 Annual Technical Conference*, Boston, MA, June 2001, pp. 35-49.



# A Mechanism for TCP-Friendly Transport-level Protocol Coordination

David E. Ott and Ketan Mayer-Patel  
University of North Carolina at Chapel Hill  
{ott,kmp}@cs.unc.edu

## Abstract

In this paper, we identify an emerging and important application class comprised of a set of processes on a cluster of devices communicating to a remote set of processes on another cluster of devices across a common intermediary Internet path. We call these applications *cluster-to-cluster applications*, or *C-to-C applications*. The networking requirements of C-to-C applications present unique challenges. Because the application involves communication between clusters of devices, very few streams will share a complete end-to-end path. At the same time, network performance needs to be measured globally across all streams for the application to employ interstream adaptation strategies. These strategies are important for the application to achieve its global objectives while at the same time realizing an aggregate flow behavior that is congestion controlled and responsive. We propose a mechanism called the *Coordination Protocol* (CP) to provide this ability. In particular, CP makes fine-grained measurements of current network conditions across all associated flows and provides transport-level protocols with aggregate available bandwidth information using an equation-based congestion control algorithm. A prototype of CP is evaluated within a network simulator and is shown to be effective.

## 1 Introduction

Advances in broadband networking, the emergence of information appliances (e.g., TiVo, PDA's, HDTV, etc.), and the now ubiquitous computer provide an environment rife with possibilities for new sophisticated multimedia applications that truly incorporate multiple media streams and interactivity. We believe many of these future Internet applications will increasingly make use of multiple communication and computing devices in

a distributed fashion. Examples of these applications include distributed sensor arrays, tele-immersion [13], computer-supported collaborative workspaces (CSCW) [7], ubiquitous computing environments [16], and complex multi-stream, multimedia presentations [17]. In these applications, no one device or computer produces or manages all of the data streams transmitted. Instead, the endpoints of communication are collections of devices. We call applications of this type *cluster-to-cluster applications*, or *C-to-C applications*.

C-to-C applications share three important properties:

- They generate many independent, but semantically related, flows of data.
- While very few flows within the application will share the exact same end-to-end path, all flows will share a common intermediary path between clusters.
- This shared common path is the primary contributor of transmission delay and the source of dynamic network conditions including loss, congestion, and jitter.

Traditional multimedia applications like streaming video generate only a few media streams (e.g., audio and video) which in general originate and terminate at the same devices (e.g., media server to media client). The applications we envision go far beyond this traditional model and include myriad flows of information of many different types communicated between clusters of devices.

Each flow of information may play a different role within the application and thus should be matched with a specific transport-level protocol which provides the appropriate end-to-end networking behavior. Furthermore, these flows will have complex semantic relationships which must be exploited by the application to appropriately adapt to changing network conditions and respond to user interaction.

*The fundamental problem with current transport-level protocols within the C-to-C application context is their lack of coordination.*

Application streams share a common intermediary path between clusters, and yet operate in isolation from one another. As a result, flows may compete with one another when network resources become limited, instead of cooperating to use available bandwidth in application-controlled ways.

In this paper, we describe and evaluate a mechanism that allows transport-level protocol coordination of separate, but semantically related, flows of data. Our approach is to introduce mechanisms at the first- and last-hop routers which make measurements of current network conditions integrated across all flows associated with a particular C-to-C application. These measurements are then communicated to the transport-level protocols on each endpoint. This enables a coordinated response to congestion across all flows that reflects application-level goals and priorities. We leverage recent work in equation-based congestion control to ensure that the aggregate bandwidth used by all of the flows is TCP-friendly while allowing the application to allocate available bandwidth to individual flows in whatever manner suits its purposes.

The main contributions of this paper are:

- Identification of the C-to-C class of Internet applications, including a brief motivating example.
- Description of the networking challenges unique to this application type.
- A proposal for a mechanism that provides transport-level protocol coordination in C-to-C applications.
- Evaluation of several aspects of our mechanism using simulation.

The rest of this paper is organized as follows: In Section 2, we present the C-to-C application model, describe a motivating example, and discuss networking requirements unique to this class of distributed applications. In Section 3, we review related work. We present our solution to the transport-level protocol coordination problem in Section 4, and provide some experimental evaluation in Section 5. Section 6 mentions future work, and Section 7 briefly summarizes the contents of this paper.

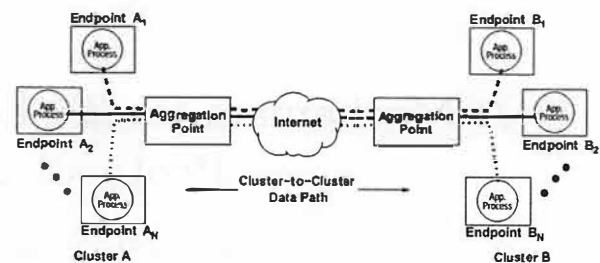


Figure 1: C-to-C application model.

## 2 Motivation

In this section, we describe in more detail the C-to-C application model, and illustrate it with a specific example. We then discuss the networking challenges associated with this application type, and why there is a need for a protocol coordination mechanism.

### 2.1 C-to-C Application Model

We model a generic C-to-C application as two sets of processes executing on two sets of communication or computing devices. Figure 1 illustrates this model.

A *cluster* is comprised of a set of *endpoints* distributed over a set of *endpoint hosts* (computers or communication devices) and a single *aggregation point*, or *AP*. Each endpoint is a process that sends and/or receives data from another endpoint belonging to a remote cluster. The AP functions as a gateway node traversed by all cluster-to-cluster flows. The common traversal path between aggregation points is known as the *C-to-C data path*.

The AP is typically the first-hop router connecting the cluster to the Internet and the cluster endpoints are typically on the same local area network. This configuration, however, is not strictly required by our model or our proposed mechanism. Our model is intended to capture several important characteristics of C-to-C applications. First, networking resources among endpoints of the same cluster are generally well provisioned for the needs of the application. Second, latency between endpoints of the same cluster is small compared to latency between endpoints on different clusters. Third, there exists a natural point within the network topology through which all cluster-to-cluster communication flows which can act as the AP. Finally, the C-to-C data path between AP's is the main source of dynamic network conditions such as jitter, congestion, and delay. Our overall objective is to coordinate endpoint flows across the C-to-C data path.



Figure 2: The Office of the Future.

## 2.2 An Example Application

A concrete example of a C-to-C application may help clarify the types of applications we envision. In the *Office of the Future*, conceived by Fuchs et al. [13], tens of digital light projectors are used to make almost every surface of an office (walls, desktops, etc.) a display surface. Similarly, tens of video cameras are used to capture the office environment from a number of different angles. At real-time rates, the video streams are used as input to stereo correlation algorithms to extract 3D geometry information. Audio is also captured from a set of microphones. The video streams, geometry information, and audio streams are all transmitted to a remote Office of the Future environment. At the remote environment, the video and audio streams are warped using both local and remote geometry information and stereo views are mapped to the light projectors. Audio is spatialized and sent to a set of speakers. Users within each Office of the Future environment wear shutter glasses that are coordinated with the light projectors.

The result is an immersive 3D experience in which the walls of one office environment essentially disappear to reveal the remote environment and provide a tele-immersive collaborative space for the participants. Furthermore, synthetic 3D models may be rendered and incorporated into both display environments as part of the shared, collaborative experience. Figure 2 is an artistic illustration of the application. A prototype of the application is described in [13].

The Office of the Future is a good example of a C-to-C application because the endpoints of the application are collections of devices. Two similarly equipped offices must exchange myriad data

streams. While few streams (if any) will share a complete end-to-end communication path, all of the data streams will span a common shared path between the local networking environments of each Office of the Future.

The local network environments are not likely to be the source of congestion, loss, or other dynamic network conditions because they can be provisioned to support the Office of the Future application. The shared Internet path between two Office of the Future environments, however, is not under local control and thus will be the source of dynamic network conditions.

The Office of the Future has a number of complex application-level adaptation strategies that we believe are typical of C-to-C applications. One such strategy, for example, is *dynamic interstream prioritization*. Since media types are integrated into a single immersive display environment, user interaction with any given media type may have implications for how other media types are encoded, transmitted, and displayed. The orientation and position of the user's head, for example, indicates a region of interest within the office environment. Media streams that are displayed within that region of interest should receive a larger share of available bandwidth and be displayed at higher resolutions and frame rates than media streams that are outside the region of interest. When congestion occurs, lower priority streams should react more strongly than higher priority streams. In this way, appropriate aggregate behavior is achieved and dynamic, application-level tradeoffs are exploited.

## 2.3 Networking Requirements of C-to-C Applications

A useful metaphor for visualizing the networking requirements of C-to-C applications is to view the communication between clusters as a rope with frayed ends. The rope represents the aggregate data flow between clusters. Each strand represents one particular flow between endpoints. At the ends of the rope, each frayed strand represents a separate path between an endpoint and its local AP. The strands come together at the AP's to form a single aggregate object. While each strand is a separate entity, they share a common fate and purpose when braided together.

With this metaphor in mind, we identify several important networking requirements of C-to-C applications:

- **Preserved end-to-end semantics.**

The transport-level protocol (i.e., TCP, UDP,

RTP, RAP, etc.) that is used by each flow is specific to the communication requirements of the data within the flow and the role it plays within the application. Thus, each transport-level protocol should maintain the appropriate end-to-end semantics and mechanisms. For example, if a data flow contains control information that requires in-order, reliable delivery, then the transport-level protocol used (e.g., TCP) should provide these services on an end-to-end basis.

- **Global coordinated measurements of throughput, delay, and loss.**

The application is interested in overall performance which may involve complex interstream adaptation strategies in the face of changing network conditions. Throughput, delay, and loss should be measured across all flows associated with the application as an aggregate. Furthermore, the behavior of individual transport-level protocols must reflect both the end-to-end semantics associated with the protocol as well as application-level adaptation strategies. To achieve this, we need to separate the adaptive dynamic behavior of each transport-level protocol from the mechanisms used to measure current network conditions.

- **TCP-friendliness.**

While the C-to-C application is free to prioritize how bandwidth is allocated among its streams, the total bandwidth used needs to be responsive to congestion. The emerging gold-standard for evaluating responsiveness is TCP-friendliness. Intuitively, a flow of data is considered TCP-friendly if it consumes as much bandwidth as a competing TCP flow consumes given the same network conditions. The advantage of using TCP-friendliness as a standard by which to measure the congestion response of a protocol (or in our case, the aggregate behavior of a set of protocols) is that it ensures "fairness" with the large majority of Internet traffic (including HTTP) that uses TCP as an underlying data transport protocol.

- **Information about peer flows.**

Individual streams within the C-to-C application may require knowledge about other streams of the same application. This knowledge can be used to determine the appropriate adaptive behavior given application-level knowledge about interstream relationships. For example, an application may want to establish a relationship between two flows of data such that one

flow consumes twice as much bandwidth as the other.

- **Flexibility for the application.**

A C-to-C application should be free to exploit trade-offs without constraint. That is, a coordination mechanism should not preclude dynamic changes in bandwidth usage among flows, or enforce any particular scheme for establishing bandwidth usage relationships between flows. The application should be free to implement whatever adaptation policy is most appropriate in whatever manner is most appropriate.

## 3 Related Work

### 3.1 Application-level Framing

The ideas of this paper are firmly grounded in the concept of Application Level Framing (ALF) [5]. The ALF principle states that networking mechanisms should be coordinated with application-level objectives. As explained above, however, C-to-C applications present unique challenges because these objectives involve interstream tradeoffs for flows that do not share a complete end-to-end path. The actions of heterogeneous protocols distributed among a cluster of devices must be coordinated to incorporate application-specific knowledge. In essence, we are extending the ALF concept to the idea of adapting protocol behavior to reflect application-level semantics. This idea is also well expressed in a position paper by Padmanabhan [11].

### 3.2 Protocol Coordination

The coordination problem presented by C-to-C applications is addressed most directly by Balakrishnan et al. in their work on the Congestion Manager (CM) [3, 1, 2]. CM provides a framework for different transport-level protocols to share information on network conditions, specifically congestion, thus allowing substantial performance improvements. We note, however, that CM flows share the same end-to-end path, while C-to-C flows share only a common intermediary path. The fact that C-to-C senders do not reside on the same host significantly limits the extensibility of the CM architecture to our problem context. CM offers applications sharing the same macroflow a system API and callback mechanisms for coordinating send events. Implementing this scheme using message passing between hosts is at best, problematic.

Furthermore, CM makes use of a scheduler to apportion bandwidth among flows. In [3], this is implemented using a Hierarchical Round Robin (HRR) algorithm. We might extend this scheme to the C-to-C context by placing the scheduler at the AP. Doing so, however, results in several problems. First, packet buffering mechanisms are required which, along with scheduling, add complexity to the AP and hurt forwarding performance. Second, packet buffering at the AP lessens endpoint control over send events since endpoint packets can be queued for an indeterminate amount of time. Balakrishnan et al. deliberately avoid buffering for exactly this reason, choosing instead to implement a scheduled callback event. Finally, scheduler configuration is problematic since C-to-C applications are complex and may continually change the manner in which aggregate bandwidth is apportioned among flow endpoints.

In [9], Kung and Wang propose a scheme for aggregating traffic between two points within a backbone network, and applying the TCP congestion control algorithm to the whole bundle. The mechanism is transparent to applications and does not provide a way for a particular application to make upstream tradeoffs.

Pradhan et al. propose a way of aggregating TCP connections sharing the same traversal path in order to share congestion control information [12]. Their scheme takes a TCP connection and divides it into two separate (“implicit”) TCP connections: a “local subconnection” and a “remote subconnection.” This scheme, however, breaks the end-to-end semantics of the transport protocol.

[14] describes a scheme for sharing congestion information across TCP flows from different hosts. This work is similar to ours in that a mechanism is introduced within the network itself to coordinate congestion response across a number of different flows which may not share a complete end-to-end path. Their mechanism does not provide the application with information about flows as an aggregate, however, and focuses on optimizing TCP performance by avoiding slow-start and detecting congestion as early as possible.

Finally, Seshan et al. propose the use of *performance servers* that act as a repository for end-to-end performance information [15]. This information may be reported by individual clients or collected by *packet capture hosts*, and then made available to client applications using a query mechanism. The time granularity of performance information is coarse compared to CP, however, since it is intended to enable smart application decisions on connection type and destination, and not ongoing congestion

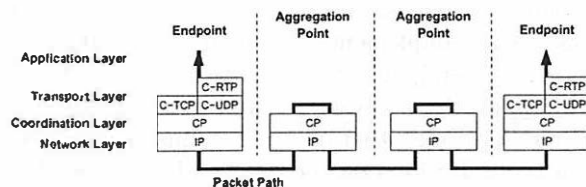


Figure 3: CP network architecture.

responsiveness. In addition, their work does not associate heterogeneous flows belonging to the same application, or consider the performance of flow aggregates.

### 3.3 Equation-based Congestion Control

TCP-friendly equation-based congestion control has recently matured as a technique for emulating TCP behavior without replicating TCP mechanics. In [6, 10], an analytical model for TCP behavior is derived that can be used to estimate the appropriate TCP-friendly rate given estimates of various channel properties. A number of important recommendations for using their TCP-friendly equation-based congestion control have been documented in [8].

## 4 Coordination Protocol (CP)

In this section we describe our solution to the problem of transport-level protocol coordination in C-to-C applications.

### 4.1 The Coordination Protocol (CP)

We propose the use of a new protocol which operates between the network layer (IP) and transport layer (TCP, UDP, etc.) that addresses the need for transport-level coordination. We call this protocol the *Coordination Protocol (CP)*. The coordination function provided by CP is transport protocol independent. At the same time, CP is distinct from network-layer protocols like IP that play a more fundamental role in routing a packet to its destination.

CP works by attaching probe information to packets transmitted from one cluster to another. As additional probe information is returned along the reverse cluster-to-cluster data path, a picture of current network conditions is formed by the AP and shared among endpoints within the local cluster. A consistent view of network conditions across flows follows from the fact that the same information is shared among all endpoints.



Figure 3 shows our proposed network architecture from a stack implementation point of view. CP exists on each endpoint device participating in the C-to-C application, as well as on the two aggregation points (APs) on either end of the cluster-to-cluster data path. Routers on the data path between APs need *not* be CP-enabled since they examine only the IP header of each incoming packet in order to route the packet in their customary manner.

The decision to insert CP between the network and transport layer rather than handling coordination at the application level requires some justification. Of primary importance to us is the preservation of end-to-end semantics. An alternative would be for each endpoint to send to a multiplexing agent who would send the data, along with probe information, to a demultiplexing agent on the remote cluster. By breaking the communication path into three stages, however, the end-to-end semantics of individual transport-level protocols have been severed. Such a scheme would also mandate that application-level control is centralized and integrated into the multiplexing agent.

Furthermore, we note that CP logically belongs between the network and transport layer. While the network layer handles the next-hop forwarding of individual packets and the transport layer handles the end-to-end semantics of individual streams, CP is concerned with streams that share a significant number of hops along the forwarding path but do not share the same end-to-end path. This relaxed notion of a stream bundle logically falls between the strict end-to-end notion of the transport-level and the independent packet notion of the network-level.

Finally, placement of CP between the network and transport layer allows for greater efficiency. In an application-level implementation of CP, information on network conditions (e.g., round trip time between APs) must pass up through an endpoint's protocol stack to the application layer. The information must then be passed back down to the transport layer where sending rate adjustments can be made in response to the information. In contrast, a distinct coordination layer allows for the information to be received and passed directly to the transport layer in a single pass as the incoming packet is processed by each layer of its endpoint's network stack.

While we acknowledge that implementing CP mechanisms at the application layer is indeed possible, we believe there are distinct advantages to the approach we have chosen. We emphasize, however, that the relative merits or drawbacks of our scheme are merely implementation issues that should not obscure the fundamental problem of C-to-C flow co-

ordination described in this paper.

## 4.2 CP Packet Headers

Figure 4 shows a CP data packet. CP encapsulates transport-level packets by prepending a 16-byte header and indicating in the protocol field which transport level protocol is associated with the packet. In turn, IP encapsulates CP packets and indicates in its protocol field that CP is being used.

Each CP header contains an application identifier associating the packet with a particular C-to-C application, and a flow identifier indicating which flow from a given endpoint host the packet belongs to. The triple (*application id*, *IP address*, *flow id*) uniquely identifies each flow within the C-to-C application, and hence the source of each CP packet. The header also contains a version number and a flags field.

The remaining contents of the CP header vary according to the changing role played by the header as it traverses the network path from source endpoint to destination endpoint. As the packet passes from the source endpoint to its local AP, the header merely identifies the cluster application it is associated with and its sender. As the packet is sent from the source's local AP to the remote AP, the header contains probe information used to measure round trip time, detect packet loss, and communicate current loss rate and bandwidth availability. As the packet is forwarded from the remote AP to its destination endpoint, the header contains information on application bandwidth use, flow membership, round trip time, loss rate, and bandwidth availability.

## 4.3 Basic Operation

The basic operation of CP is as follows.

- **As packets originate from source endpoints.**

The CP header is included in the application packet indicating the source of the packet and the cluster application it is associated with.

- **As packets arrive at the local AP.**

CP will process the identification information arriving in the CP header, and note the packet's size and arrival time. Part of the CP header will then be overwritten, allowing the AP to communicate congestion probe information to the remote AP.

- **As packets arrive at the remote AP.**

The CP header is processed and used to detect network conditions. Again, part of the CP

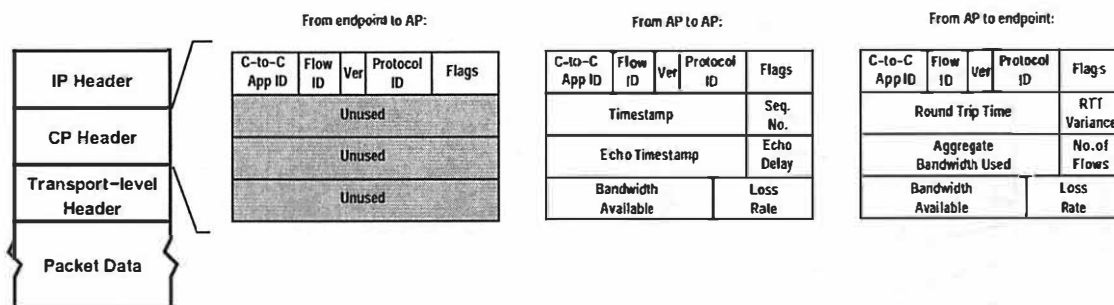


Figure 4: CP packet structure.

header is overwritten to communicate network condition information, along with information on cluster application size and bandwidth usage, to the remote remote endpoint.

- **As packets arrive at the destination endpoint.**

CP processes network condition information from the CP header and passes it on to the transport-level protocol and the application.

#### 4.4 State Maintained by an AP

An AP maintains a table of active cluster applications, each entry of which exists as soft state. When a packet arrives with an unknown cluster identifier in its CP header, a new entry will be created in the table and CP probe mechanisms will become active for that application. Similarly, if no CP packet has been seen for a particular cluster identifier  $i$ , then the entry will time out and be removed from the application table. Use of soft state in this manner is both flexible and lightweight in that it avoids the need for explicit configuration and ongoing administration.

For each cluster application, the AP monitors the number of participating flows, and the number and size of packets received during a given interval. Weighted averages are calculated to dampen the effect of packet bursts. The information is passed back to local cluster endpoints using the CP header whenever a packet arrives from the remote AP on route to a local endpoint. If no such packet arrives within a specified time period, then a *report packet* is created and “pushed” to each endpoint informing them of cluster application membership and bandwidth usage, as well as current network conditions.

An AP also maintains probe state, including a current packet sequence number, estimated round trip time and mean deviation, a loss history and estimated loss rate, and a bandwidth availability calculation. Use of these mechanisms is described below.

#### 4.5 Detecting Network Delay and Loss

A primary function of CP is to measure network delay and detect packet loss along the cluster-to-cluster data path. Figure 5, Table 1, and Table 2 together illustrate how information in the CP header is used to make these measurements.

Each packet passing from one AP to another has several numbers inserted into its CP header. The first is a sequence number that increases monotonically for every packet sent. A remote AP may use this number to observe gaps (and reorderings) in the aggregate flow of cluster application packets that it receives. In this way, it can detect losses and infer congestion. In our example, AP2 detects the loss of packet C when the sequence number received skips from 14 (packet A) to 16 (packet D).

In addition, a timestamp is sent along with the sequence number indicating the time at which the AP sent the packet. The remote AP will then echo the timestamp of the last sequence number received by placing the value in the CP header of the next packet traveling on the reverse path back to the sending AP. Along with this timestamp, a delay value will also be given indicating the length of time between the arrival of the sequence number at the AP and the time the AP transmitted the echo.

By noting the time when a packet is received ( $T_{arrival}$ ), the AP can calculate the round trip time as  $(T_{arrival} - T_{echo}) - T_{delay}$ . In our example, AP2 receives packet B at time 280. The CP header contains the timestamp echo 60 and an echo delay value of 30. Thus, the round trip time is calculated as  $280 - 60 - 30 = 190$ . A weighted average of these round trip time calculations is used to dampen the effects of burstiness.

Note that because sequence numbers in the CP header do not have any transport-level function, CP can use whatever C-to-C application packet is being transmitted next to carry this information. Since

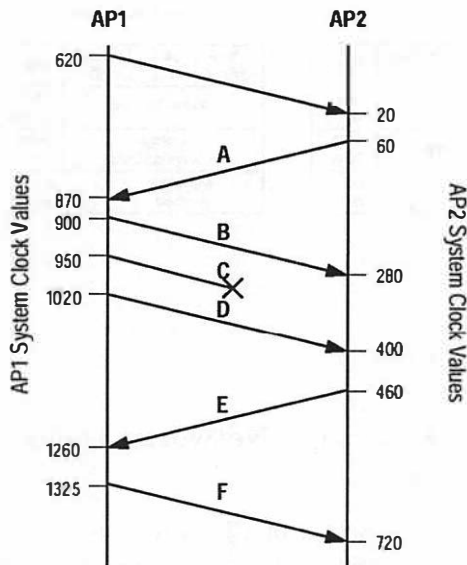


Figure 5: Timeline of AP packet exchanges.

Packet	Sequence Number	Time-stamp	Time-stamp Echo	Echo Delay
B	14	900	60	30
C	15	950	60	80
D	16	1020	60	150
F	17	1325	460	65

Table 1: Information in CP header for packets traveling from AP1 to AP2 in Figure 5.

the packets of multiple flows are available for this purpose, this mechanism can be used for fine-grained detection of network conditions along the cluster-to-cluster data path.

We also observe that there is no one-to-one correspondence between timestamps sent and timestamps echoed between APs. It may be the case that more than one packet is received by a remote AP before a packet traveling along the opposite path is available to echo the most current timestamp. The AP simply makes use of available packets in a best effort manner. In Figure 5 this can be seen as AP2 receives both packets B and D before packet E is available to send on the return path. Likewise, an AP may echo the same timestamp more than once if no new CP packet arrives with a new timestamp. In our example, this occurs when AP1 sends packets B, C, and D with a timestamp echo value of 60 which it received from packet A.

Packet	Sequence Number	Time-stamp	Time-stamp Echo	Echo Delay
A	76	60	620	40
E	77	460	1020	60

Table 2: Information in CP header for packets traveling from AP2 to AP1 in Figure 5.

#### 4.6 Calculating Loss Rate and Bandwidth Availability

Calculation of loss rate and bandwidth availability make use of equation-based congestion control methods described in Floyd et al. in their work on TCP-friendly rate control (TFRC) [8].

Loss rate, a central input parameter into the bandwidth availability equation, is calculated using a *loss history* and *loss events* rather than individual packet losses. By using a loss event rate rather than a simple lost packet rate, we provide a more stable handling of lost packet bursts. The reader is referred to [6] for more details.

Calculation of available bandwidth makes use of the equation:

$$X = \frac{s}{R\sqrt{\frac{2bp}{3}} + t_{RTO}(3\sqrt{\frac{3bp}{8}})p(1 + 32p^2)} \quad (1)$$

where  $X$  is the transmit rate (bytes/sec),  $s$  is the packet size (bytes),  $R$  is the round trip time (sec),  $p$  is the loss event rate on the interval  $[0,1.0]$ ,  $t_{RTO}$  is the TCP retransmission timeout (sec), and  $b$  is the number of packets acknowledged by a single TCP acknowledgement.

The resulting quantity, which we refer to as current *bandwidth availability*, is calculated at the remote AP, and then passed using the CP header to each endpoint in the cluster. Similarly, the event loss rate is also passed on to endpoints to inform them of current network conditions.

We emphasize here that the use of the above equation to calculate bandwidth availability for the cluster application makes the *aggregate* data flow from one AP to another TCP-compatible.

#### 4.7 Transport-level Protocols

Transport-level protocols at the endpoints are built on top of CP in the same manner that TCP is built on top of IP. CP provides these transport-level protocols with a consistent view of network conditions, including aggregate bandwidth availability, loss rate, and round trip delay measurements. In addition, it

informs endpoints of the aggregate bandwidth usage and the current number of flows in the cluster application. A transport-level protocol will in turn use this information, along with various configuration parameters, to determine a data transmission rate and related send characteristics.

In Figure 3, we show several possible transport-level protocols (C-TCP, C-UDP, and C-RTP) which are meant to represent coordinated counterparts to existing protocols. A coordinated version of UDP (C-UDP) simply makes the above information available directly to the application which may modify its sending rate according to an application-specific rule or bandwidth sharing scheme.

A coordinated version of TCP (C-TCP) may consider acknowledgements only as an indicator of successful transfer. The burden of round trip delay determination and congestion detection can be relegated entirely to CP. Send rate adjustments at the transport level are the combined result of configuration information given by the application (e.g., a maximum sending rate), and information on current network conditions as provided by CP.

While C-UDP and C-TCP represent adaptations of familiar transport-level protocols, we believe that other coordinated transport-level protocols are possible. Such protocols will make use of CP information and application semantics to adjust sending rates to meet application-specific objectives.

## 4.8 Application-level Programming Interface

Endpoint implementations of CP provide a modified socket interface to the application layer. With this interface, the application is able to associate its data flow with a particular cluster application and interact more directly with CP-related mechanisms in two ways.

First, the application may use the interface to communicate configuration information to the transport-level. For example, an application may wish to restrict the transport-level sending rate to no more than some maximum value. Or, an application may instruct the transport layer to send at only some fraction of the available bandwidth given various conditions. Such configuration is made possible by a set of system calls which allow applications to pass functions to the transport layer which operate on reported CP values in order to calculate an instantaneous sending rate.

The application may also use the interface to access CP information directly. Thus, a system call is provided which allows the application to query, for

example, available bandwidth, round trip time, and the current loss rate. Obtaining this information directly is of particular importance when the application itself controls its own send rate (e.g., C-UDP) rather than relegating such control to the transport-level protocol (e.g., C-TCP).

## 4.9 Endpoint Coordination

While a goal of C-to-C applications is to maintain congestion responsiveness on an aggregate level, how this goal is realized is left entirely to the application. The approach of CP is to avoid the use of traffic shaping or packet scheduling mechanisms at the AP, but instead to provide application endpoints with bandwidth availability “hints” and other information about changing network conditions. An application may then apportion bandwidth among endpoints by configuring them to respond to these hints in ways which meet the objectives of the application as a whole.

For example, a C-to-C application may configure secondary streaming endpoints to reduce their sending rate, or stop sending altogether, in response to a drop in available bandwidth below a particular threshold value. At the same time, a primary stream endpoint may continue to send at its original rate, and a control endpoint may increase its sending rate somewhat in order to transmit important commands telling the receive side how to respond to the change. Despite these differences in response behavior, the aggregate bandwidth usage drops appropriately to match the bandwidth availability hint given.

CP provides a C-to-C application with the mechanisms needed to make coordinated adaptation decisions which reflect the current state of the network and the application’s objectives. We believe it unnecessary to provide additional mechanisms which enforce bandwidth usage among endpoints since each belongs to the same application and thus shares the same objectives. In addition, endpoint configuration may be complex and change dynamically making the implementation of an enforcement scheme inherently problematic.

## 5 Evaluation

In this section, we evaluate the behavior of CP using the network simulator *ns-2* [4]. We focus here on our implementation of C-TCP, the coordinated counterpart to TCP.

C-TCP, like TCP, implements reliability through the use of acknowledgement packets, timeouts, and

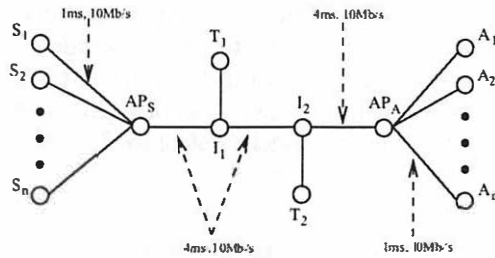


Figure 6: Simulation testbed in ns2.

retransmission. Unlike window-based TCP, however, C-TCP is a rate-based implementation which adjusts its instantaneous send rate based on bandwidth availability information supplied by CP, and configuration information supplied by the application. Our implementation of C-TCP draws heavily from TFRC [8], except that loss and send rate calculations are handled by APs communicating over the C-to-C data path, and TCP-compatibility, as defined in [6], is achieved on an aggregate and not per-flow level.

## 5.1 Network Topology

Our simulation topology is pictured in Figure 6. A cluster of sending agents is labeled  $S_1$  through  $S_n$ , with its local aggregation point labeled  $AP_S$ . A remote cluster of ACK (acknowledgement) agents is labeled  $A_1$  through  $A_n$ , with its aggregation point labeled  $AP_A$ .  $I_1$  and  $I_2$  are intermediary nodes used to create a congested link, and  $T_1$  and  $T_2$  are used for traffic generation.

Propagation delay on links  $AP_S-I_1$ ,  $I_1-I_2$ , and  $I_2-AP_A$  is configured to be 4 msec, while it is only 1 msec on links  $S_i-AP_S$  and  $A_i-AP_A$ . The link capacity for all links is 10 Mb/s, except for links  $T_1-I_1$  and  $T_2-I_2$  where link capacity is 100 Mb/s. This allows traffic generators to increase traffic over link  $I_1-I_2$  to any desired level.

Trace data is collected as it is transmitted from  $AP_S$  to  $I_1$  since this allows us to observe sending rates before additional traffic on the link  $I_1-I_2$  causes queuing delays, drops, or jitter not reflective of cluster endpoint sending rates.

TCP and C-TCP flows in this section use an infinitely large data source and send at the maximum rate allowed by their respective algorithms. Congestion periods are created by configuring  $T_1$  and  $T_2$  to generate constant bitrate traffic across the link  $I_1-I_2$ . In particular, a CBR agent sending at a constant 7.5-9.0 Mb/s from  $T_1$  to  $T_2$  competes with data traffic from  $S_1-S_n$  over link  $I_1-I_2$ .

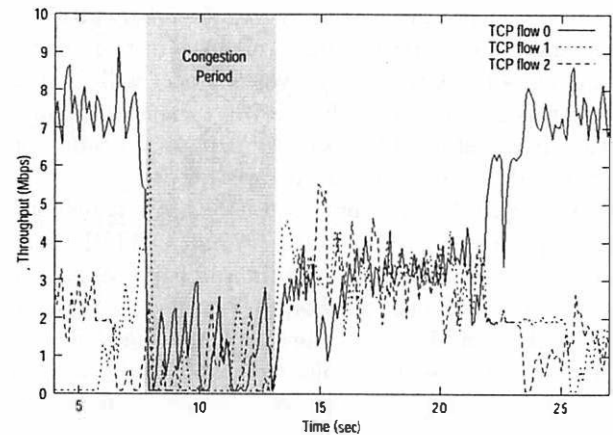


Figure 7: TCP flows competing for bandwidth during congestion.

## 5.2 Behavior of Uncoordinated TCP Flows

To better see the problem addressed by CP, we first examine how several TCP connections behave without coordination. In Figure 7, we see the throughput plot of three TCP connections as network congestion occurs between time 8.0 and 13.0 seconds. Flow 0 belongs to an application process with higher bandwidth requirements than processes associated with flows 1 and 2. This can be seen clearly at the right and left edges of the plot when flow 0 takes its full share of the bandwidth under congestion-free circumstances.

We note the following observations:

- During the congestion interval, all three flows compete with one another and receive a roughly similar portion of the available bandwidth.
- The flows continue to compete in a similar fashion during the period directly afterward (time 13.0 through 22.0) as each struggles to send accumulated data and regain its requisite level of bandwidth.
- The bandwidth used by each flow is characterized by jagged edges, often criss-crossing one another. This makes sense since each flow operates independently, searching the bandwidth space by repeatedly ramping up and backing off.

## 5.3 Behavior of C-TCP Flows

We postulate here that use of the Coordination Protocol (CP) should be distinctive in at least two ways. First, since all flows make use of the same bandwidth availability calculation, round trip time,



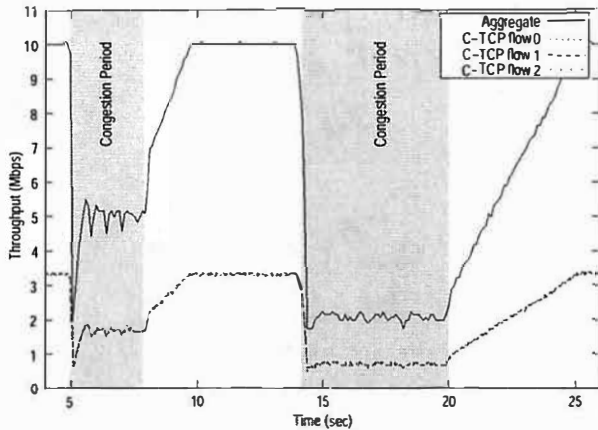


Figure 8: C-TCP flows sharing bandwidth equally.

and loss rate information, bandwidth usage patterns among CP flows should be much smoother. That is, there should be far fewer jagged edges and less criss-crossing of individual flow bandwidths as flows need not search the bandwidth space in isolation for a maximal send rate.

Second, the use of bandwidth by a set of CP flows should reflect the priorities and configuration of the application—including intervals of congestion when network resources become limited.

To test these hypotheses, we implemented three simple bandwidth sharing schemes which reflect different objectives an application may wish to achieve on an aggregate level. We note here that more schemes are possible, and the mixing of schemes in complex, application-specific ways is an open area of research.

Figure 8 shows a simple *equal bandwidth* sharing scheme in which C-TCP flows divide available bandwidth ( $B$ ) equally among themselves. ( $R_i = B/N$  where  $R_i$  is the send rate for sending endpoint  $i$ , and  $N$  is the number of sending endpoints.) The aggregate plot line shows the total bandwidth used by the multi-flow application at a given time instant. While not plotted on the same graph, this line closely corresponds to bandwidth availability values calculated by APs and communicated to cluster endpoints.

Figure 8 confirms our hypothesis that usage patterns among CP flows should be far smoother, and avoid the jagged criss-crossing effect seen in Figure 7. This is both because flows are not constantly trying to ramp up in search of a maximal sending rate, and because of the use of weighted averages in the bandwidth availability calculation itself. The latter has the effect of dampening jumps in value from one instant to the next.

Figure 9 shows a *proportional bandwidth* shar-

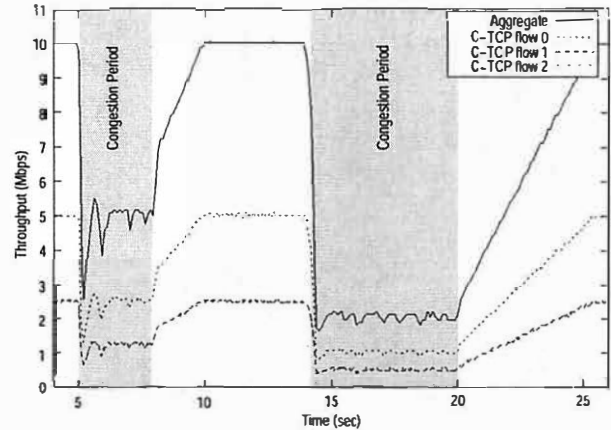


Figure 9: C-TCP flows sharing bandwidth proportionally.

ing scheme among C-TCP flows. In this particular scheme, flow 0 is configured to take .5 of the bandwidth ( $R_0 = .5 * B$ ), while flows 1 and 2 evenly divide the remaining portion for a value of .25 each ( $R_1 = R_2 = .25 * B$ ).

Figure 9 confirms our second hypothesis above by showing sustained proportional sharing throughout the entire time interval. This includes the congestion intervals (times 5.0-8.0 and 14.0-20.0) and post-congestion intervals (times 8.0-10.0, 20.0-25.0) when TCP connections might still contend for bandwidth.

In Figure 10, we see a *constant bandwidth* flow in conjunction with two flows equally sharing the remaining bandwidth. The former is configured to send at a constant rate of 3.5 Mb/s or, if it is not available, at the bandwidth availability value for that given instant. ( $R_0 = \min(3.5 \text{ Mb/s}, B)$ ). Flows 1 and 2 split the remaining bandwidth or, if none is available, send at a minimum rate of 1Kb/s. ( $R_1 = R_2 = \max((B - R_0)/2, 1 \text{ Kb/s})$ )

We observe that flows 1 and 2 back off their sending rate almost entirely whenever flow 0 does not receive its full share of bandwidth. We also note that while flow 0 is configured to send at a constant rate, it never exceeds available bandwidth limitations during time of congestion.

We emphasize once again the impossibility of achieving results like Figure 9 and Figure 10 in an application without the transport-level coordination provided by CP.

## 5.4 TCP-Friendliness

The TCP-friendliness of aggregate CP traffic is established by using the equation-based congestion control method described in [6] and used by

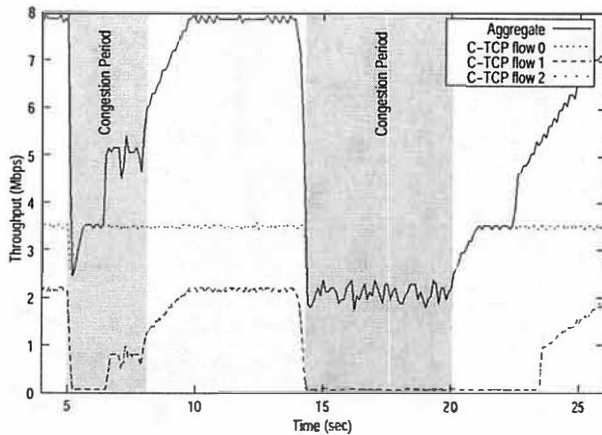


Figure 10: A constant bandwidth C-TCP flow with two C-TCP flows sharing remaining bandwidth.

TFRC [8].

While equation-based rate control guarantees TCP-compatibility over long time intervals, Figure 11 illustrates informally the behavior of a single C-TCP connection with two TCP connections during a short congested interval (time 5.0 through 9.0). Here we're interested in verifying that the behavior of the C-TCP flow does indeed appear to be compatible with that of the TCP flows.

In general, we see that the C-TCP connection mixes reasonably well with the TCP connections, receiving approximately an equal share of the available bandwidth. In addition, we once again observe the smoothness of its rate adjustments compared to the far more volatile changes in TCP flows.

## 6 Future Work

We believe transport-level protocol coordination in C-to-C applications to be fertile area for future work. In particular, much work remains to be done on new transport protocols better equipped to make use of network condition and cluster flow information. These protocols may provide end-to-end semantics which are more specific to an application's needs than current all-purpose protocols like TCP and UDP.

Flow coordination in a C-to-C application within this paper has meant the sharing of bandwidth from a single bandwidth availability calculation, equivalent to a single TCP-compatible flow. Future work might focus on sharing the equivalent of more than one TCP-compatible flow, just as many applications (eg., Web browsers) open more than one connection to increase throughput by parallelizing end-to-end

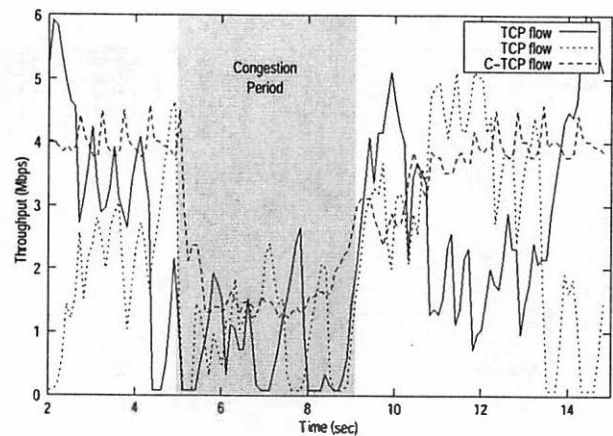


Figure 11: C-TCP flow interacting with TCP flows.

communication.

The assumption that local networks on each end of a C-to-C application can always be provisioned to minimize network delay and loss may not always be true. For example, wireless devices may introduce delay and loss inherent to the technology itself. How CP can be adapted to accommodate this situation is an area of future work. One idea is to use CP for distinguishing between congestion sources. End-to-end estimates of delay and loss could be compared with those of CP in order to determine whether congestion is local or within the network.

Finally, the impact of CP mechanisms on forwarding performance at the AP is an important issue that deserves further study. We conjecture here that the impact will be modest since per-packet processing largely amounts to simple accounting and checksum computations, and an AP avoids entirely the need for buffering or scheduling mechanisms. An actual implementation is required, however, before any meaningful analysis can be done.

## 7 Summary

In this paper, we have identified a class of distributed applications known as *cluster-to-cluster (C-to-C) applications*. Such applications have semantically related flows that share a common intermediary path, typically between first- and last-hop routers. C-to-C applications require transport-level coordination to better put the application in control over bandwidth usage, especially during periods when network resources become limited by congestion. Without coordination, high-priority flows may contend equally with low-priority flows for bandwidth, or receive no bandwidth at all, thus preventing the application from meeting its objectives entirely.

We have proposed the Coordination Protocol

(CP) as a way of coordinating semantically related flows in application-controlled ways. CP operates between the network (IP) and transport (TCP, UDP) layers, offering C-to-C flows fine-grained information about network conditions along the cluster-to-cluster data path, as well as information about application flows as an aggregate. In particular, CP makes use of equation-based rate control methods to calculate bandwidth availability for the entire C-to-C application. This results in aggregate flow rates that are highly adaptive to changing network conditions and TCP-compatible.

## References

- [1] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. System Support for Bandwidth Management and Content Adaptation in Internet Applications. *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 213–226, October 2000.
- [2] H. Balakrishnan and S. Seshan. *RFC 3124: The Congestion Manager*, June 2001.
- [3] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An Integrated Congestion Management Architecture for Internet Hosts. *Proceedings of ACM SIGCOMM*, September 1999.
- [4] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in Network Simulation. *IEEE Computer*, 33(5):59–67, May 2000.
- [5] D.D. Clark and D.L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. *Proc. ACM SIGCOMM 1990, Computer Communication Review*, 20(4):200–208, September 1990.
- [6] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-Based Congestion Control for Unicast Applications. *Proceedings of ACM SIGCOMM*, pages 43–56, 2000.
- [7] J. Grudin. Computer-Supported Cooperative Work: Its History and Participation. *Computer*, 27(4):19–26, 1994.
- [8] M. Handley, J. Padhye, S. Floyd, and J. Widmer. *TCP Friendly Rate Control (TFRC): Protocol Specification*. IETF, May 2001. Internet Draft, work in progress.
- [9] H.T. Kung and S.Y. Wang. TCP Trunking: Design, Implementation and Performance. *Proc. of ICNP '99*, November 1999.
- [10] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and Its Empirical Validation. *Proceedings of ACM SIGCOMM*, 1998.
- [11] V.N. Padmanabhan. Coordinated Congestion Management and Bandwidth Sharing for Heterogeneous Data Streams. *Proceedings of the 9th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 187–190, 1999.
- [12] P. Pradhan, T. Chiueh, and A. Neogi. Aggregate TCP Congestion Control Using Multiple Network Probing. *Proc. of IEEE ICDCS 2000*, 2000.
- [13] Ramesh Raskar, Greg Welch, Matt Cutts, Adam Lake, Lev Stessin, and Henry Fuchs. The Office of the Future: A Unified Approach to Image-Based Modeling and Spatially Immersive Displays. *Proceedings of ACM SIGGRAPH 98*, 1998.
- [14] S. Savage, N. Cardwell, and T. Anderson. The Case for Informed Transport Protocols. *Proceedings of HotOS VII*, March 1999.
- [15] Srinivasan Seshan, Mark Stemm, and Randy H. Katz. SPAND: Shared Passive Network Performance Discovery. In *USENIX Symposium on Internet Technologies and Systems*, 1997.
- [16] M. Weiser. Some Computer Science Problems in Ubiquitous Computing. *Communications of the ACM*, 36(7):75–84, July 1993.
- [17] T.-P. Yu, D. Wu, K. Mayer-Patel, and L.A. Rowe. DC: A Live Webcast Control System. *Proc. of SPIE Multimedia Computing and Networking*, 2001.



# My cache or yours? Making storage more exclusive

Theodore M. Wong

Carnegie Mellon University, Pittsburgh, PA

tmwong+@cs.cmu.edu

John Wilkes

Hewlett-Packard Laboratories, Palo Alto, CA

wilkes@hpl.hp.com

## Abstract

Modern high-end disk arrays often have several gigabytes of cache RAM. Unfortunately, most array caches use management policies which duplicate the same data blocks at both the client and array levels of the cache hierarchy: they are *inclusive*. Thus, the aggregate cache behaves as if it was only as big as the larger of the client and array caches, instead of as large as the sum of the two. Inclusiveness is wasteful: cache RAM is expensive.

We explore the benefits of a simple scheme to achieve *exclusive caching*, in which a data block is cached at either a client or the disk array, but not both. Exclusiveness helps to create the effect of a single, large unified cache. We introduce a DEMOTE operation to transfer data ejected from the client to the array, and explore its effectiveness with simulation studies. We quantify the benefits and overheads of demotions across both synthetic and real-life workloads. The results show that we can obtain useful—sometimes substantial—speedups.

During our investigation, we also developed some new cache-insertion algorithms that show promise for multi-client systems, and report on some of their properties.

## 1 Introduction

Disk arrays use significant amounts of cache RAM to improve performance by allowing asynchronous read-ahead and write-behind, and by holding a pool of data that can be re-read quickly by clients. Since the per-gigabyte cost of RAM is much higher than of disk, cache can represent a significant portion of the cost of modern arrays. Our goal here is to see how best to exploit it.

The cache sizes needed to accomplish read-ahead and write-behind are typically tiny compared to the disk capacity of the array. Read-ahead can be efficiently handled with buffers whose size is only a few times the track size of the disks. Write-behind can be handled with buffers whose size is large enough to cover the variance (burstiness) in the write workload [32, 39], since the sustained average transfer rate is bounded by what the disks can support—everything eventually has to get to stable

storage. Overwrites in the write-behind cache can increase the front-end write traffic supported by the array, but do not intrinsically increase the size of cache needed.

Unfortunately, there is no such simple bound for the size of the re-read cache: in general, the larger the cache, the greater the benefit, until some point of diminishing returns is reached. The common rule of thumb is to try to cache about 10% of the active data. Table 1 suggests that this is a luxury out of reach of even the most aggressive cache configurations if all the stored data were to be active. Fortunately, this is not usually the case: a study of UNIX file system workloads [31] showed that the mean working set over a 24 hour period was only 3–7% of the total storage capacity, and the 90th percentile working set was only 6–16%. A study of deployed HP AutoRAID systems [43] found that the working set rarely exceeded the space available for RAID1 storage (about 10% of the total storage capacity).

Both array and client re-read caches are typically operated using the *least-recently-used* (LRU) cache replacement policy [11, 12, 35]; even though many proprietary tweaks are used in array caches, the underlying algorithm is basically LRU [4]. Similar approaches are the norm in client-server file system environments [15, 27].

Interactions between the LRU policies at the client and array cause the combined caches to be *inclusive*: the array (lower-level) cache duplicates data blocks held in the client (upper-level) cache, so that the array cache is pro-

High-end arrays		
System	Cache	Disk space
EMC 8830	64 GiB	70 TB
IBM ESS	32 GiB	27 TB
HP XP512	32 GiB	92 TB

High-end servers		
System	Memory	Type (CPUs)
IBM z900	64 GiB	High-end (1–16)
Sun E10000	64 GiB	High-end (4–64)
HP Superdome	128 GiB	High-end (8–64)
HP rp8400	64 GiB	Mid-range (2–16)
HP rp7400	32 GiB	Mid-range (2–8)

Table 1: Some representative maximum-supported sizes for disk arrays and servers from early 2002. 1 GiB =  $2^{30}$  bytes.



viding little re-read benefit until it exceeds the effective size of the client caches.

Inclusiveness is wasteful: it renders a chunk of the array cache similar in size to the client caches almost useless. READ operations that miss in the client are more likely to miss in the array and incur a disk access penalty. For example, suppose we have a client with 16 GB of cache memory connected to a disk array with 16 GB of re-read cache, and suppose the workload has a total READ working set size of 32 GB. (This single client, single array case is quite common in high-end computer installations; with multiple clients, the effective client cache size is equal to the amount of unique data that the clients caches hold, and the same arguments apply.) We might naïvely expect the 32 GB of available memory to capture almost all of the re-read traffic, but in practice it would capture only about half of it, because the array cache will duplicate blocks that are already in the client [15, 27].

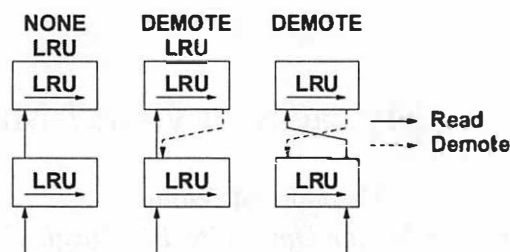
To avoid these difficulties, it would be better to arrange for the combined client and array caches to be *exclusive*, so that data in one cache is not duplicated in the other.

## 1.1 Exclusive caching

Achieving exclusive caching requires that the client and array caches be managed as one. Since accesses to the client cache are essentially free, while accesses to the array cache incur the round-trip network delay, the cost of an I/O operation at the client, and the controller overheads at the array, we can think of this setup as a cache hierarchy, with the array cache at the lower level. These costs are not large: modern storage area networks (SANs) provide 1–2 Gbit/s of bandwidth per link, and I/O overheads of a few hundred microseconds; thus, retrieving a 4 KB data block can take as little as 0.2 ms.

However, it would be impractical to rewrite client O/S and array software to explicitly manage both caches. It would also be undesirable for the array to keep track of precisely which blocks are in the client, since this metadata is expensive to maintain. However, we can approximate the desired behavior by arranging that the client (1) tells the array when it changes what it caches, and (2) returns data ejected from the upper-level cache to the lower-level one, rather than simply discarding it.

We achieve the desired behavior by introducing a DEMOTE operation, which one can think of as a possible extension to the SCSI command set. DEMOTE works as follows: when a client is about to eject a clean block from its cache (e.g., to make space for a READ), it first tries to return the block to the array using a DEMOTE. A DEMOTE operation is similar to a WRITE operation: the array tries to put the demoted block into its re-read cache, ejecting another block if necessary to make space.



**Figure 1:** Sample cache management schemes. The top and bottom boxes represent the client and array cache replacement queues respectively. The arrow in a box points to the end closest to being discarded.

Unlike a WRITE, the array short-circuits the operation (i.e., it does not transfer the data) if it already has a copy of the block cached, or if it cannot immediately make space for it. In all cases, the client then discards the block from its own cache.

Clients are trusted to return the same data that they read earlier. This is not a security issue, since they could easily issue a WRITE to the same block to change its contents. If corruption is considered a problem, the array could keep a cryptographic hash of the block and compare it with a hash of the demoted block, at the expense of more metadata management and execution time.

SANs are fast and disks are slow, so though a DEMOTE may incur a SAN block transfer, performance gains are still possible: even small reductions in the array cache miss rate can achieve dramatic reductions in the mean READ latency. Our goal is to evaluate how close we can get to this desirable state of affairs and the benefits we obtain from it.

## 1.2 Exclusive caching schemes

The addition of a DEMOTE operation does not in itself yield exclusive caching: we also need to decide what the array cache does with blocks that have just been demoted or read from disk. This is primarily a choice of cache replacement policy. We consider three combinations of demotions with different replacement policy at the array, illustrated in figure 1; all use the LRU policy at the client:

- NONE-LRU (the baseline scheme): clients do no demotions; the array uses the LRU replacement policy for both demoted and recently read blocks.
- DEMOTE-LRU: clients do demotions; the array uses the traditional LRU cache management for both demoted and recently read blocks.
- DEMOTE: clients do demotions; the array puts blocks it has sent to a client at the head (closest to

being discarded end) of its LRU queue, and puts demoted blocks at the tail. This scheme most closely approximates the effect of a single unified LRU cache.

We observe that the DEMOTE scheme is more exclusive than the DEMOTE-LRU scheme, and so should result in lower mean latencies. Consider what happens when a client READ misses in the client and array caches, and thus provokes a back-end disk read. With DEMOTE-LRU, the client and array will double-cache the block until enough subsequent READs miss and push it out of one of the caches (which will take at least as many READs as the smaller of the client and array queue lengths). With DEMOTE, the double-caching will only last until the next READ that misses in the array cache. We thus expect DEMOTE to be more exclusive than DEMOTE-LRU, and so to result in lower mean READ latencies.

### 1.3 Objectives

To evaluate the performance of our exclusive caching approach, we aim to answer the following questions:

1. Do demotions increase array cache hit rates in single-client systems?
2. If so, what is the overall effect of demotions on mean latency? In particular, do the costs exceed the benefits? Costs include extra SAN transfers, as well as delays incurred by READs that wait for DEMOTES to finish before proceeding.
3. How sensitive are the results to variations in SAN bandwidth?
4. How sensitive are the results to the relative sizes of the client and array caches?
5. Do demotions help when an array has multiple clients?

The remainder of the paper is structured as follows. We begin with a demonstration of the potential benefits of exclusive caching using some simple examples. We then explore how well it fares on more realistic workloads captured from real systems, and show that DEMOTE does indeed achieve the hoped-for benefits.

Multi-client exclusive caching represents a more challenging target, and we devote the remainder of the paper to an exploration of how this can be achieved—including a new way of thinking about cache insertion policies. After surveying related work, we end with our observations and conclusions.

## 2 Why exclusive caching?

In this section, we explore the *potential* benefits of exclusive caching in single-client systems, using a simple analytical performance model. We show that exclusive caching has the potential to double the effective cache size with client and array caches of equal size, and that the potential speedups merit further investigation.

We begin with a simple performance model for estimating the costs and benefits of caching. We predict the mean latency seen by a client application as

$$T_{mean} = T_c h_c + (T_a + T_c) h_a + (T_a + T_c + T_d) miss \quad (1)$$

where  $T_c$  and  $T_a$  are costs of a hit in the client and disk array caches respectively,  $T_d$  is the cost of reading a block from disk (since such a block is first read into the cache, and then accessed from there, it also incurs  $T_a + T_c$ ),  $h_c$  and  $h_a$  are the client and array cache hit rates respectively (expressed as fractions of the total client READs), and  $miss = 1 - (h_c + h_a)$  is the miss rate (the fraction of all READs that must access the disk). Since  $T_c \approx 0$ ,

$$T_{mean} \approx T_a h_a + (T_a + T_d) miss \quad (2)$$

In practice,  $T_a$  is much less than  $T_d$ :  $T_a \approx 0.2$  ms and  $T_d \approx 4$ –10 ms for non-sequential 4 KB reads.

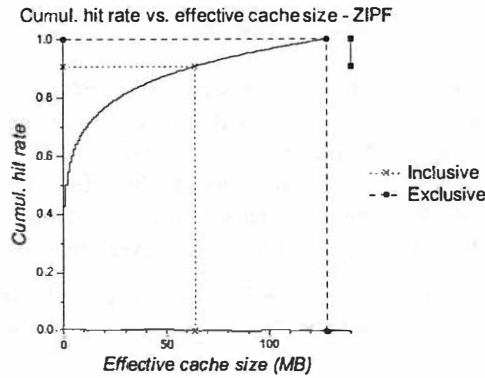
We must also account for the cost of demotions. Large demotions will be dominated by data transfer times, small ones by array controller and host overheads. If we assume that a DEMOTE costs the same as a READ that hits in the array, and that clients demote a block for every READ, then we can approximate the cost of demotions by doubling the latency of array hits. This is an upper bound, since demotions transfer no data if they abort, e.g., if the array already has the data cached. With the inclusion of demotion costs,

$$T_{mean} \approx 2T_a h_a + (2T_a + T_d) miss. \quad (3)$$

We now use our model to explore some simple examples, setting  $T_a = 0.2$  ms and  $T_d = 10$  ms throughout this section.

### 2.1 Random workloads

Consider first a workload with a spatially uniform distribution of requests across some working set (also known as *random*). We expect that a client large enough to hold



**Figure 2:** Cumulative hit rate vs. effective cache size for a Zipf-like workload, with client and array caches of 64 MB each and a working set size of 128 MB. The marker shows the additional array hit rate achieved with exclusive caching.

half of the working set would achieve  $h_c = 50\%$ . An array with inclusive caching duplicates the client contents, and would achieve no additional hits, while an array with exclusive caching should achieve  $h_a = 50\%$ .

Equations 2 and 3 predict that the change from inclusive to exclusive caching would reduce the mean latency from  $0.5(T_a + T_d)$  to  $T_a$ , i.e., from 5.1 ms to 0.2 ms.

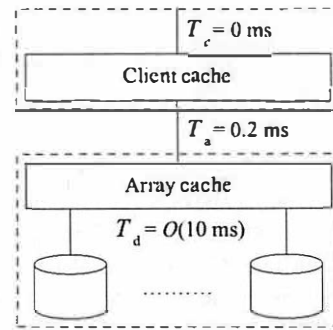
## 2.2 Zipf workloads

Even workloads that achieve high client hit rates may benefit from exclusive caching. An example of such a workload is one with a Zipf-like distribution [49], which approximates many common access patterns: a few blocks are frequently accessed, others much less often. This is formalized as setting the probability of a READ for the  $i^{\text{th}}$  block proportional to  $1/i^\alpha$ , where  $\alpha$  is a scaling constant commonly set to 1.

Consider the cumulative hit rate vs. effective cache size graph shown in figure 2 for the Zipf workload with a 128 MB working set. A client with a 64 MB cache will achieve  $h_c = 91\%$ . No additional hits would occur in the array with a 64 MB cache and traditional, fully inclusive caching. Exclusive caching would allow the same array to achieve an incremental  $h_a = 9\%$ ; because  $T_d \gg T_a$ , even small decreases in the miss rate can yield large speedups. Equations 2 and 3 predict mean READ latencies of 0.918 ms and 0.036 ms for inclusive and exclusive caching respectively—an impressive  $25.5\times$  speedup.

## 3 Single-client synthetic workloads

In this section, we explore the effects of exclusive caching using simulation experiments with synthetic workloads. Our goal is to confirm the intuitive arguments presented in section 2, as well as to conduct sen-



**Figure 3:** System simulated for the single-client workloads, with a RAID5 array and a 1 Gbit/s FibreChannel SAN.

sitivity analyses for how our demotion scheme responds to variations in the client-array SAN bandwidth and relative client and array cache sizes. Sections 4 and 5 present our results for real-life workloads.

### 3.1 Evaluation environment: Pantheon

To evaluate our cache management schemes, we began by using the Pantheon simulator [44], which includes calibrated disk models [33]. Although the Pantheon array models have not been explicitly calibrated, Pantheon has been used successfully in design studies of the HP AutoRAID disk array [45], so we have confidence in its predictive powers.

We configured Pantheon to model a RAID5 disk array connected to a single client over a 1 Gbit/s FibreChannel link, as shown in figure 3. For these experiments, we used a workload with 4 KB READs, and set  $T_a = 0.2$  ms; the Pantheon disk models gave  $T_d \approx 10$  ms.

The Pantheon cache models are extremely detailed, keeping track of I/O operations in 256 byte size units in order to model contention effects. Unfortunately, this requires large amounts of memory, and restricted us to experiments with only 64 MB caches. With a 4 KB cache block size, this means that the client and array caches were restricted to  $N_c = N_a = 16384$  blocks in size.

To eliminate resource-contention effects for our synthetic workload results, we finished each READ before starting the next. In each experiment, we first “warmed up” the caches with a working-set size set of READs; the performance of these READs is not included in the results. Latency variances were all below 1%.

Our chief metric for evaluating the exclusive caching schemes is the mean latency of a READ at the client; we also report on the array cache hit rate. For each result, we present both absolute latencies and a *speedup* ratio, which is the baseline (NONE-LRU) mean latency divided by the mean latency for the current experiment. Although the difficulties of modeling partially closed-loop

Workload	Client	NONE-LRU	DEMOTÉ-LRU	DEMOTÉ
RANDOM	50%	8%	21%	46%
SEQ	0%	0%	0%	100%
ZIPF	86%	2%	4%	9%

**Table 2:** Client and array cache hit rates for single-client synthetic workloads. The client hit rates are the same for all the demotion variants, and can be added to the array hit rates to get the total cache hit rates.

Workload	NONE-LRU	DEMOTÉ-LRU	DEMOTÉ
RANDOM	4.77 ms	3.43 ms (1.39×)	0.64 ms (7.5×)
SEQ	1.67 ms	1.91 ms (0.87×)	0.48 ms (3.5×)
ZIPF	1.41 ms	1.19 ms (1.18×)	0.85 ms (1.7×)

**Table 3:** Mean READ latencies and speedups over NONE-LRU for single-client synthetic workloads.

application behavior are considerable [16], a purely I/O-bound workload should see its execution time reduced by the speedup ratio.

### 3.2 The RANDOM synthetic workload

For this test, the workload consisted of one-block READs uniformly selected from a working set of  $N_{rand}$  blocks. Such random access patterns are common in on-line transaction-processing workloads (e.g., TPC-C, a classic OLTP benchmark [38]).

We set the working set size to the sum of the client and array cache sizes:  $N_c = N_a = 16384$ ,  $N_{rand} = 32768$  blocks, and issued  $N_{rand}$  warm-up READs, followed by  $10 \times N_{rand}$  timed READs.

We expected that the client would achieve  $h_c = 50\%$ . Inclusive caching would result in no cache hits at the array, while exclusive caching should achieve an additional  $h_a = 50\%$ , yielding a dramatic improvement in mean latency.

The results in table 2 validate our expectations. The client achieved a 50% hit rate for both inclusive and exclusive caching, and the array with DEMOTÉ achieved an additional 46% hit rate. 4% of READs still missed with DEMOTÉ, because the warm-up READs did completely fill the client cache. Also, since NONE-LRU is not fully inclusive (as previous studies demonstrate [15]), the array with NONE-LRU still achieved an 8% hit rate.

As predicted in section 1.2, DEMOTÉ-LRU did not perform as well as DEMOTÉ. DEMOTÉ-LRU only achieved  $h_a = 21\%$ , while DEMOTÉ achieved  $h_a = 46\%$ , which was a 7.5× speedup over NONE-LRU, as seen in table 3.

Figure 4 compares the cumulative latencies achieved with NONE-LRU and DEMOTÉ. For DEMOTÉ, the jump at 0.4 ms corresponds to the cost of an array hit plus the

cost of a demotion. In contrast, NONE-LRU got fewer array hits (table 2), and its curve has a significantly smaller jump at 0.2 ms, which is the cost of an array cache hit without a demotion.

### 3.3 The SEQ synthetic workload

Sequential accesses are common in scientific, decision-support and data-mining workloads. To evaluate the benefit of exclusive caching for such workloads, we simulated READs of sequential blocks from a working set of  $N_{seq}$  contiguous blocks, chosen so that the working set would fully occupy the combined client and array caches:  $N_c = N_a = 16384$ , and  $N_{seq} = N_c + N_a - 1 = 32767$  blocks (the  $-1$  accounts for double-caching of the most recently read block). We issued  $N_{seq}$  warm-up READs, followed by  $10 \times N_{seq}$  timed one-block READs.

We expected that at the end of the warm-up period, the client would contain the blocks in the second half of the sequence, and an array under exclusive caching would contain the blocks in the first half. Thus, with DEMOTÉ, all subsequent READs should hit in the array. On the other hand, with NONE-LRU and DEMOTÉ-LRU, we expected that the array would always contain the same blocks as the client; neither the client nor the array would have the next block in the sequence, and all READs would miss.

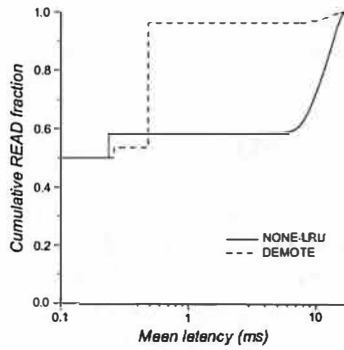
Again, the results in table 2 validate our expectations. Although no READs ever hit in the client, they all hit in the array with DEMOTÉ. The mean latency for DEMOTÉ-LRU was higher than for NONE-LRU because it pointlessly demoted blocks that the array discarded before they were reused. Although all READs missed in both caches with NONE-LRU and DEMOTÉ-LRU, the mean latencies of 1.67 ms and 1.91 ms respectively were less than the random-access disk latency  $T_d$  thanks to read-ahead in the disk drive [33].

The cumulative latency graph in figure 4 further demonstrates the benefit of DEMOTÉ over NONE-LRU: all READs with DEMOTÉ had a latency of 0.4 ms (the cost of an array hit plus a demotion), while all READs with NONE-LRU had latencies between 1.03 ms (the cost of a disk access with read-ahead caching) and 10 ms (the disk latency  $T_d$ , incurred when the READ sequence wraps around). Overall, DEMOTÉ achieved a 3.5× speedup over NONE-LRU, as seen in table 3.

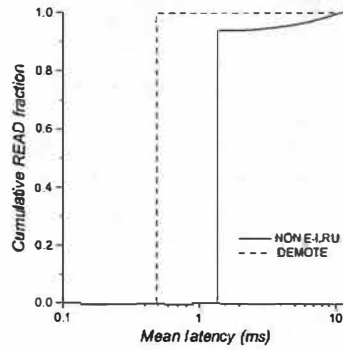
### 3.4 The ZIPF synthetic workload

Our Zipf workload sent READs from a set of  $N_{zipf}$  blocks, with  $N_{zipf} = 1.5(N_c + N_a)$ , so for  $N_c = N_a = 16384$ ,  $N_{zipf} = 49152$ . This resulted in three equal

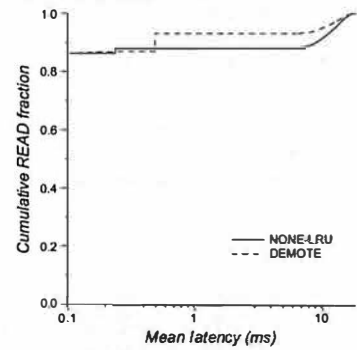
Cumul. READ frac. vs. mean latency - RANDOM



Cumul. READ frac. vs. mean latency - SEQ



Cumul. READ frac. vs. mean latency - ZIPF



**Figure 4:** Cumulative READ fraction vs. mean READ latency for the RANDOM, SEQ, and ZIPF workloads with NONE-LRU and DEMOTE.

size sets of  $N_{\text{Zipf}}/3$  blocks:  $Z_0$  for the most active third (which received 90% of the accesses),  $Z_1$  for the next most active (6% of the accesses), and  $Z_2$  for the least active (the remaining 4% of the accesses). We issued  $N_{\text{Zipf}}$  warm-up READs, followed by  $10 \times N_{\text{Zipf}}$  timed READs.

We expected that at the end of the warm-up set, the client cache would be mostly filled with blocks from  $Z_0$  with the highest request probabilities, and that an array under exclusive caching would be mostly filled with the blocks from  $Z_1$  with the next highest probabilities. With our test workload, exclusive caching schemes should thus achieve  $h_c = 90\%$  and  $h_a = 6\%$  in steady state. On the other hand, the more inclusive caching schemes (NONE-LRU and DEMOTE-LRU) would simply populate the array cache with the most-recently read blocks, which would be mostly from  $Z_0$ , and thus achieve a lower array hit rate  $h_a$ .

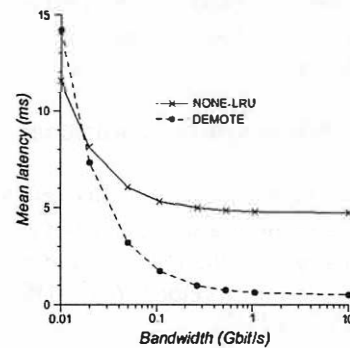
The results in table 2 validate our expectations. The client always achieved  $h_c = 86\%$  (slightly lower than the anticipated 90% due to an incomplete warm-up). But there was a big difference in  $h_a$ : DEMOTE achieved 9%, while NONE-LRU achieved only 2%.

The cumulative latency graph in figure 4 supports this: as with RANDOM, the curve for DEMOTE has a much larger jump at 0.4 ms (the cost of an array hit plus a demotion) than NONE-LRU does at 0.2 ms (the cost of an array hit alone). Overall, DEMOTE achieved a  $1.7\times$  speedup over NONE-LRU, as seen in table 3. This may seem surprising given the modest increase in array hit rate, but is more readily understandable when viewed as a decrease in the overall miss rate from 12% to 5%.

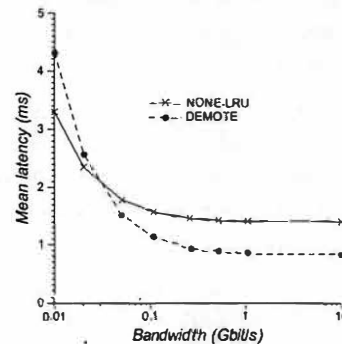
### 3.5 SAN bandwidth sensitivity analysis

Exclusive caching using demotions relies on a low-latency, high-bandwidth SAN to allow the array cache to perform as a low-latency extension of the client cache.

Mean latency vs. bandwidth - RANDOM



Mean latency vs. bandwidth - ZIPF



**Figure 5:** Mean READ latency vs. SAN bandwidth for the RANDOM and ZIPF workloads.

The more this expectation is violated (i.e., as SAN latency increases), the less benefit we expect to see—possibly to the point where demotions are not worth doing. To explore this effect, we conducted a sensitivity analysis, using Pantheon to explore the effects of varying the simulated SAN bandwidth from 10 Gbit/s to 10 Mbit/s on the NONE-LRU and DEMOTE schemes.

Our experiments validated our expectations. Figure 5 shows that at very low effective SAN bandwidths (less



than 20–30 Mbit/s), NONE-LRU outperformed DEMOTE, but DEMOTE won as soon as the bandwidth rose above this threshold. The results for RANDOM and ZIPF are similar, except that the gap between the NONE-LRU and DEMOTE curves for high-bandwidth networks is smaller for ZIPF since the increase in array hit rate (and the resultant speedup) was smaller.

### 3.6 Evaluation environment: *fscachesim*

For subsequent experiments, we required a simulator capable of modeling multi-gigabyte caches, which was beyond the abilities of Pantheon. To this end, we developed a simulator called *fscachesim* that only tracks the client and array cache contents, omitting detailed disk and SAN latency measurements. *fscachesim* is simpler than Pantheon, but its predictive effects for our study are similar: we repeated the experiments described in sections 3.2 and 3.4 with identical workloads, and confirmed that the client and array hit rates matched exactly. We used *fscachesim* for all the experimental work described in the remainder of this paper.

### 3.7 Cache size sensitivity analysis

In the results reported so far, we have assumed that the client cache is the same size as the array cache. This section reports on what happens if we relax this assumption, using a 64 MB client cache and RANDOM and ZIPF.

We expected that an array with the NONE-LRU inclusive scheme would provide no reduction in mean latency until its cache size exceeds that of the client, while one with the DEMOTE exclusive scheme would provide reductions in mean latency for any cache size until the working set fits in the aggregate of the client and array caches.

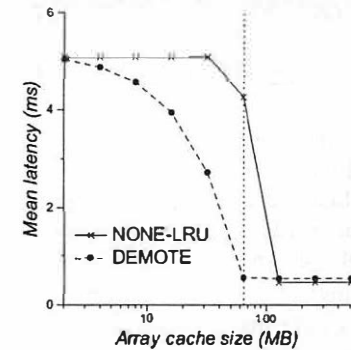
The results in figure 6 confirm our expectations. Maximum benefit occurs when the two caches are of equal size, but DEMOTE provides benefits over roughly a 10:1 ratio of cache sizes on either side of the equal-size case.

### 3.8 Summary

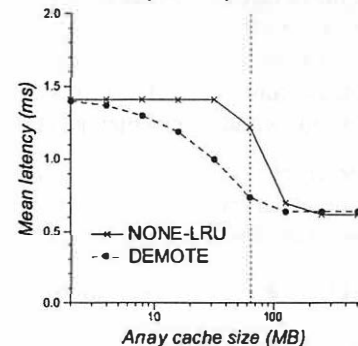
The synthetic workload results show that DEMOTE offers significant potential benefits: 1.7–7.5× speedups are hard to ignore. Better yet, these benefits are mostly insensitive to variations in SAN bandwidth and only moderately sensitive to the client:array cache size ratio.

Since our results showed that DEMOTE-LRU never outperformed DEMOTE, we did not consider it further. We also investigated schemes with different combinations of LRU and *most-recently-used* (MRU) replacement policies at the client and array in conjunction with demotions, and found that none performed as well as DEMOTE.

Mean latency vs. array cache size - RANDOM



Mean latency vs. array cache size - ZIPF



**Figure 6:** Mean READ latency vs. array cache size for the RANDOM and ZIPF workloads. The client cache size was fixed at 64 MB. The 64 MB size is marked with a dotted line.

Workload	Date	Capacity	Cache	Clients	Length	Warm-up	I/Os
CELLO99	1999	300 GB	2 GB	1	1 month	1 day	61.9 M
DB2	—	5.2 GB	—	8	2.2 hours	30 min	3.7 M
HTTPD	1995	0.5 GB	—	7	24 hours	1 hr	1.1 M
OPENMAIL	1999	4260 GB	2 GB	6	1 hour	10 min	5.2 M
TPC-H	2000	2100 GB	32 GB	1	1 hour	10 min	7.0 M

**Table 4:** Real-life workload data, with date, storage capacity, array cache size, client count, trace duration, and I/O count. ‘Warm-up’ is the fraction of the trace used to pre-load the caches in our experiments. For DB2 and HTTPD, working set size instead of capacity is shown. ‘—’ are unknown entries.

## 4 Single-client real-life workloads

Having demonstrated the benefits of demotion-based exclusive caching for synthetic workloads, we now evaluate its benefits for real-life workloads, in the form of traces taken from the running systems shown in table 4.

Some of the traces available to us are somewhat old, and cache sizes considered impressive then are small today. Given this, we set the cache sizes in our experiments commensurate with the time-frame and scale of the system from which the traces were taken.

We used *fscachesim* to simulate a system model similar to the one in figure 3, with cache sizes scaled to reflect the data in table 4. We used equations 2 and 3

Workload	Client	NONE-LRU	DEMOTÉ
CELLO99	54%	1%	2.34 ms (1.28x)
DB2	4%	0%	5.01 ms (1.40x)
HTTPD	86%	3%	0.53 ms (2.20x)

**Table 5:** Client and array hit rates and mean latencies for single-client real-life workloads. Client hit rates are the same for all schemes. Latencies are computed using equations 2 and 3 with  $T_a = 0.2$  ms and  $T_d = 5$  ms. Speedups for DEMOTÉ over NONE-LRU are also shown.

with  $T_a = 0.2$  ms,  $T_d = 5$  ms to convert cache hit rates into mean latency predictions. This disk latency is more aggressive than that obtained from Pantheon, to reflect the improvements in disk performance seen in the more recent systems. We further assumed that there was sufficient SAN bandwidth to avoid contention, and set the cost of an aborted demotion to 0.16 ms (the cost of SAN controller overheads without an actual data transfer).

As before, our chief metric of evaluation is the improvement in the mean latency of a READ achieved by demotion-based exclusive caching schemes.

#### 4.1 The CELLO99 real-life workload

The CELLO99 workload comprises a trace of every disk I/O access for the month of April 1999 from an HP 9000 K570 server with 4 CPUs, about 2 GB of main memory, two HP AutoRAID arrays and 18 directly connected disk drives. The system ran a general time-sharing load under HP-UX 10.20; it is the successor to the CELLO system Ruemmler and Wilkes describe in their analysis of UNIX disk access patterns [32]. In our experiments, we simulated 2 GB client and array caches.

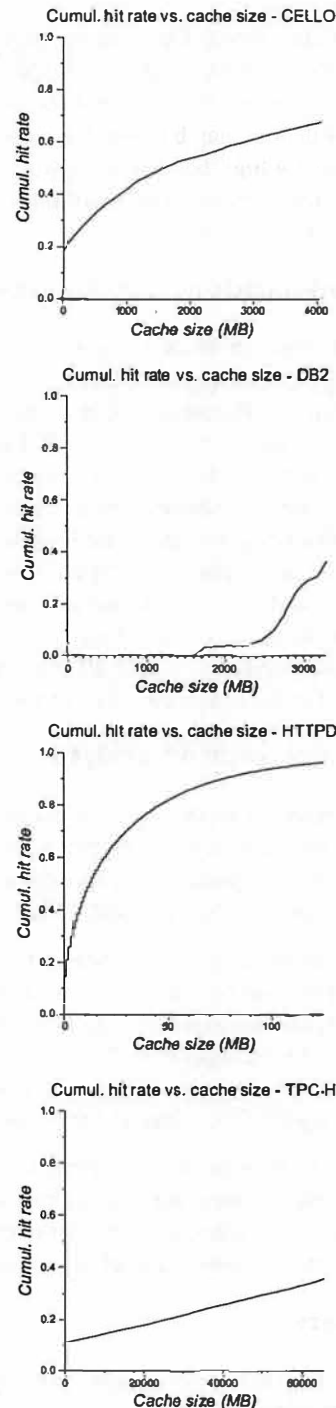
Figure 7 suggests that that switching from inclusive to exclusive caching, with the consequent doubling of effective cache size from 2 GB to 4 GB, should yield a noticeable increase in array hit rate. The results shown in table 5 demonstrate this: using DEMOTÉ achieved  $h_a = 13\%$  (compared to  $h_a = 1\%$  with NONE-LRU), yielding a 1.28x speedup—solely from changing the way the array cache is managed.

#### 4.2 The DB2 real-life workload

The DB2 trace-based workload was generated by an eight-node IBM SP2 system running an IBM DB2 database application that performed join, set and aggregation operations on a 5.2 GB data set. Uysal *et al.* used this trace in their study of I/O on parallel machines [40].

The eight client nodes accessed disjoint sections of the database; for the single-client workload experiment we combined all these access streams into one.

DB2 exhibits a behavior between the sequential and random workload styles seen in the SEQ and RANDOM syn-



**Figure 7:** Cumulative hit rate vs. cache size graphs for single-client real-life workloads.

thetic workloads. The graph for DB2 in figure 7 suggests that a single 4 GB cache would achieve about a 37% hit rate, but that a split cache with 2 GB at each of the client and array would achieve almost no hits at all with inclusive caching; thus, DEMOTÉ should do much better than NONE-LRU. The results shown in table 5 bear this out: DEMOTÉ achieved a 33% array hit rate, and a 1.40x

Array size	Client	NONE-LRU		DEMOTÉ	
2 GB	23%	0%	4.01 ms	1%	4.13 ms (0.97×)
16 GB	23%	0%	4.01 ms (1.00×)	6%	3.86 ms (1.04×)
32 GB	23%	1%	3.97 ms (1.01×)	13%	3.54 ms (1.13×)

**Table 6:** Client and array hit rates and mean latencies for single-client TPC-H for different array caches. Client hit rates and cache sizes (32 GB) are the same for all schemes. Latencies are computed using equations 2 and 3 with  $T_a = 0.2$  ms and  $T_d = 5$  ms. Speedups are with respect to a 2 GB array cache with NONE-LRU.

speedup over NONE-LRU.

### 4.3 The HTTPD real-life workload

The HTTPD workload was generated by a seven-node IBM SP2 parallel web server [22] serving a 524 MB data set. Uysal *et al.* also used this trace in their study [40]. Again, we combined the client streams into one.

HTTPD has similar characteristics to ZIPF. A single 256 MB cache would hold the entire active working set; we elected to perform the experiment with 128 MB of cache split equally between the client and the array in order to obtain more interesting results. An aggregate cache of this size should achieve  $h_c + h_a \approx 95\%$  according to the graph in figure 7, with the client achieving  $h_c \approx 85\%$ , and an array under exclusive caching the remaining  $h_a \approx 10\%$ .

Table 5 shows that the expected benefit indeed occurs: DEMOTÉ achieved a 10% array hit rate, and an impressive 2.2× speedup over NONE-LRU.

### 4.4 The TPC-H real-life workload

The TPC-H workload is a 1-hour portion of a 39-hour trace of a system that performed an audited run [18] of the TPC-H database benchmark [37]. This system illustrates high-end commercial decision-support systems: it comprised an 8-CPU (550MHz PA-RISC) HP 9000 N4000 server with 32 GB of main memory and 2.1 TB of storage capacity, on 124 disks spread across 3 arrays (with 1.6 GB of aggregate cache) and 4 non-redundant disk trays. The host computer was already at its maximum-memory configuration in these tests, so adding additional host memory was not an option. Given that this was a decision-support system, we expected to find a great deal of sequential traffic, and relatively little cache reuse. Our expectations are borne out by the results.

In our TPC-H experiments, we used a 16 KB block size, a 32 GB client cache, and a 2 GB array cache as the baseline, and explored the effects of changing the array cache size up to 32 GB. Table 6 shows the results.

The traditional, inclusive caching scheme showed no im-

provement in latency until the array cache size reached 32 GB, at which point we saw a tiny (1%) improvement.

With a 2 GB array cache, DEMOTÉ yielded a slight slowdown (0.97× speedup), because it paid the cost of doing demotions without increasing the array cache hit rate significantly. However, DEMOTÉ obtained a 1.04× speedup at 16 GB, and a 1.13× speedup at 32 GB, while the inclusive caching scheme showed no benefits. This data confirms that cache reuse was not a major factor in this workload, but indicates that the exclusive caching scheme took advantage of what reuse there was.

### 4.5 Summary

The results from real-life workloads support our earlier conclusions: apart from the TPC-H baseline, which experienced a small 0.97× slowdown due to the cost of non-beneficial demotions, we achieved up to a 2.20× speedup.

We find these results quite gratifying, given that extensive previous research on cache systems enthusiastically reports performance improvements of a few percent (e.g., a  $\sim 1.12\times$  speedup).

## 5 Multi-client systems

Multi-client systems introduce a new complication: the sharing of data between clients. Note that we are deliberately not trying to achieve client-memory sharing, in the style of protocols such as GMS [13, 42]. One benefit is that our scheme does not need to maintain a directory of which clients are caching which blocks.

Having multiple clients cache the same block does not itself raise problems (we assume that the clients wish to access the data, or they would not have read it), but exploiting the array cache as a shared resource does: it may no longer be a good idea to discard a recently read block from the array cache as soon as it has been sent to a client. To help reason about this, we consider two boundary cases here. Of course, real workloads show behavior between these extremes.

*Disjoint workloads:* The clients each issue READs for non-overlapped parts of the aggregate working set. The READs appear to the array as if one client had issued them, from a cache as large as the aggregate of the client caches. To determine if exclusive caching will help, we use the cumulative hit rate vs. cache size graph to estimate the array hit rate as if a single client had issued all READs, as in section 2.

*Conjoint workloads:* The clients issue exactly the same READ requests in the same order at the exact same time. If we arbitrarily designate the first client to issue an I/O

as the leader, and the others as followers, we see that READs that hit in the leader also will hit in the followers. The READs appear to the array as if one client had issued them from a cache as large as an individual client cache.

To determine if the leader will benefit from exclusive caching, we use the cumulative hit rate vs. cache size graph to estimate the array hit rate as if the leader had issued all READs, as in section 2.

To determine if the followers will benefit from exclusive caching, we observe that all READs that miss for the leader in the array will also cause the followers to stall, waiting for that block to be read into the array cache. As soon as it arrives there, it will be sent to the leader, and then all the followers, before it is discarded. That is, the followers will see the same performance as the leader.

In systems that employ demotion, the followers waste time demoting blocks that the leader has already demoted. Fortunately, these demotions will be relatively cheap because they need not transfer any data.

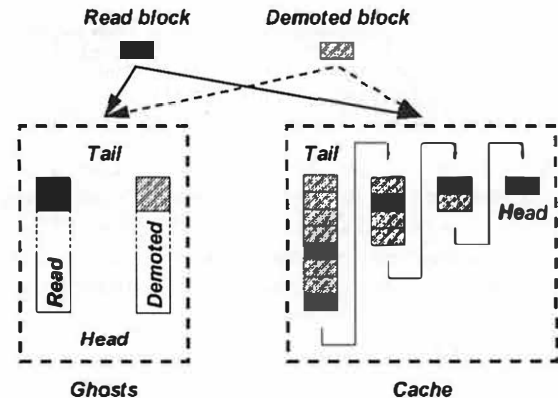
## 5.1 Adaptive cache insertion policies

Our initial results using the simple demotion-based exclusive caching scheme described above to multi-client systems were mixed. At first, we evaluated NONE-LRU and DEMOTE in a multi-client system similar to the one shown in figure 3, with the single client shown in that figure simply replaced by  $N$  clients, each with  $1/N$  of the cache memory of the single client. As expected, workloads in which clients shared few or no blocks (disjoint workloads) benefitted from DEMOTE.

Unfortunately, workloads in which clients shared blocks performed worse with DEMOTE than with NONE-LRU, because shared workloads are not conjoint in practice: clients do not typically READ the same blocks in the same order at the same time. Instead, a READ for block  $X$  by one client may be followed by several READs for other blocks before a second READ for  $X$  by another client. Recall that with DEMOTE the array puts blocks read from disk at the head of the LRU queue, i.e., in MRU order. Thus, the array is likely to eject  $X$  before the READ from the later client.

We made an early design decision to avoid the complexities of schemes that require the array to track which clients had which blocks and request copies back from them—we wanted to keep the client-to-array interaction as simple, and as close to standard SCSI, as possible.

Our first insight was that the array should reserve a portion of its cache to keep blocks recently read from disk “for a while”, in case another client requests them. To achieve this, we experimented with a segmented LRU (SLRU) array cache [21]—one with *probationary* and



**Figure 8:** Operation of read and demoted ghost caches in conjunction with the array cache. The array inserts the metadata of incoming read (demoted) blocks into the corresponding ghost, and the data into the cache. The cache is divided into segments of either uniform or exponentially-growing size. The array selects the segment into which to insert the incoming read (demoted) block based on the hit count in the corresponding ghost.

*protected* segments, each managed in LRU fashion. The array puts newly inserted blocks (read and demoted) at the tail of the probationary segment, and moves them to the tail of the protected segment if a subsequent READ hits them. The array moves blocks from the head of the protected segment to the tail of the probationary one, and ejects blocks from the head of the probationary segment.

SLRU improved performance somewhat, but the optimal size of the protected segment varied greatly with the workload: the best size was either very small (less than 8% of the total), or quite large (over 50%). These results were less robust than we desired.

Our second insight is that the array can treat the LRU queue as a continuum, rather than as a pair of segments: inserting a block near the head causes that block to have a shorter expected lifetime in the queue than inserting it near the tail. We can then use different insertion points for demoted blocks and disk-read blocks. (Pure DEMOTE is an extreme instance that only uses the ends of the LRU queue, and SLRU is an instance where the insertion point is a fixed distance down the LRU queue.)

Our experience with SLRU suggested that the array should select the insertion points *adaptively* in response to workload characteristics instead of selecting them statically. For example, the array should insert demoted blocks closer to the tail of its LRU queue than disk-read blocks if subsequent READs hit demoted blocks more often. To support this, we implemented *ghost caches* at the array for demoted and disk-read blocks.

A ghost cache behaves like a real cache except that it only keeps cache metadata, enabling it to simulate

the behavior of a real cache using much less memory. We used a pair of ghost caches to simulate the performance of hypothetical array caches that only inserted blocks from a particular source—either demotions or disk reads. Just like the real cache, each ghost cache was updated on READs to track hits and execute its LRU policy.

We used the ghost caches to provide information about which insertion sources are the more likely to insert blocks that are productive to cache, and hence where in the real cache future insertions from this source should go, as shown in figure 8.) This was done by calculating the insertion point in the real cache from the relative hit counts of the ghost caches. To do so, we assigned the value 0 to represent the head of the real array LRU queue, and the value 1 to the tail; the insertion points for demoted and disk-read blocks were given by the ratio of the hit rates seen by their respective ghost caches to the total hit rate across all ghost caches.

To make insertion at an arbitrary point more computationally tractable, we approximated this by dividing the real array LRU queue into a fixed number of segments  $N_{segs}$  (10 in our experiments), multiplying the calculated insertion point by  $N_{segs}$ , and inserting the block at the tail of that segment.

We experimented with uniform segments, and with exponential segments (each segment was twice the size of the preceding one, the smallest being at the head of the array LRU queue). The same segment-index calculation was used for both schemes, causing the scheme with segments of exponential size to give significantly shorter lifetimes to blocks predicted to be less popular.

We designated the combination of demotions with ghost caches and uniform segments at the array as DEMOTE-ADAPT-UNI, and that of demotions with ghost caches and exponential segments as DEMOTE-ADAPT-EXP. We then re-ran the experiments for which we had data for multiple clients, but separated out the individual clients.

## 5.2 The multi-client DB2 workload

We used the same DB2 workload described in section 4.2, but with the eight clients kept separate. Each client had a 256 MB cache, so the aggregate of client caches remained at 2 GB. The array had 2 GB of cache.

Each DB2 client accesses disjoint parts of the database. Given our qualitative analysis of disjoint workloads, and the speedup for DB2 in a single-client system with DEMOTE, we expected to obtain speedups in this multi-client system. If we assume that each client uses one eighth (256 MB) of the array cache, then each client has an aggregate of 512 MB to hold its part of the database,

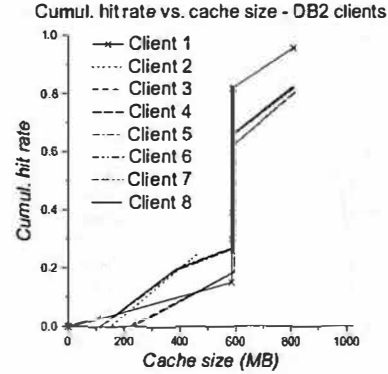


Figure 9: Cumulative hit rate vs. cache size for DB2 clients.

Client	1	2	3	4	5	6	7	8
NONE-LRU								
Mean lat.	5.20	4.00	4.62	4.66	4.66	4.68	4.66	4.66
DEMOTÉ (mean speedup 1.50x)								
Mean lat.	1.30	4.12	3.44	3.41	3.39	3.38	3.40	3.38
Speedup	4.00x	0.97x	1.34x	1.37x	1.38x	1.39x	1.37x	1.38x
DEMOTÉ-ADAPT-UNI (mean speedup 1.27x)								
Mean lat.	2.15	4.12	3.57	3.53	4.09	4.07	4.07	4.05
Speedup	2.42x	0.97x	1.29x	1.32x	1.14x	1.15x	1.15x	1.15x
DEMOTÉ-ADAPT-EXP (mean speedup 1.32x)								
Mean lat.	1.79	4.12	3.55	3.51	3.94	4.05	3.92	3.99
Speedup	2.91x	0.97x	1.30x	1.33x	1.18x	1.16x	1.19x	1.17x

Table 7: Per-client mean latencies (in ms) for multi-client DB2. Latencies are computed using equations 2 and 3 with  $T_a = 0.2$  ms and  $T_d = 5$  ms. Speedups over NONE-LRU, and the geometric mean of all client speedups, are also shown.

and we expected from figure 9 that exclusive caching would obtain a significant increase in array hit rates, with a corresponding reduction in mean latency.

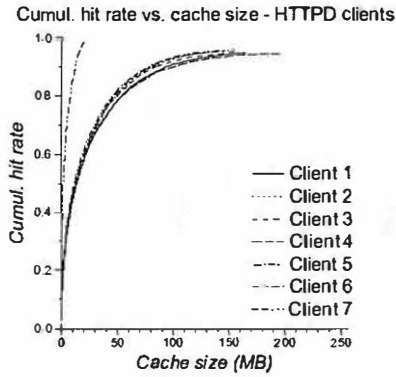
Our results shown in table 7 agree: DEMOTE achieved an impressive 1.50x speedup over NONE-LRU. DEMOTE-ADAPT-UNI and DEMOTE-ADAPT-EXP achieved only 1.27–1.32x speedups, since they were more likely to keep disk-read blocks in the cache, reducing the cache available for demoted blocks, and thus making the cache less effective for this workload.

## 5.3 The multi-client HTTPD workload

We returned to the original HTTPD workload, and separated the original clients. We gave 8 MB to each client cache, and kept the 64 MB array cache as before.

Figure 10 indicates that the per-client workloads are somewhat similar to the ZIPF synthetic workload. As shown in section 3.4, disk-read blocks for such workloads will in general have low probabilities of being reused, while demoted blocks will have higher probabilities. On the other hand, as shown by the histogram in table 8, clients share a high proportion of blocks, and tend to exhibit conjoint workload behavior. Thus, while the array should discard disk-read blocks more quickly





**Figure 10:** Cumulative hit rate vs. cache size for HTTPD clients.

No. clients	1	2	3	4	5	6	7
No. blocks	13173	8282	5371	5570	6934	24251	5280
% of total	19%	12%	8%	8%	10%	35%	8%

**Table 8:** Histogram showing the number of blocks shared by  $x$  HTTPD clients, where  $x$  ranges from 1 to 7 clients.

Client	1	2	3	4	5	6	7
NONE-LRU							
Mean lat.	0.90	0.83	0.82	0.89	0.79	0.76	0.19
DEMOTÉ (mean slowdown 0.55x)							
Mean lat.	1.50	1.41	1.44	1.48	1.43	1.33	0.46
Speedup	0.60x	0.59x	0.57x	0.60x	0.55x	0.57x	0.41x
DEMOTÉ-ADAPT-UNI (mean slowdown 0.91x)							
Mean lat.	0.99	0.92	0.91	0.98	0.87	0.86	0.20
Speedup	0.91x	0.90x	0.90x	0.91x	0.90x	0.89x	0.94x
DEMOTÉ-ADAPT-EXP (mean speedup 1.18x)							
Mean lat.	0.81	0.73	0.74	0.79	0.68	0.67	0.12
Speedup	1.12x	1.13x	1.10x	1.13x	1.16x	1.13x	1.52x

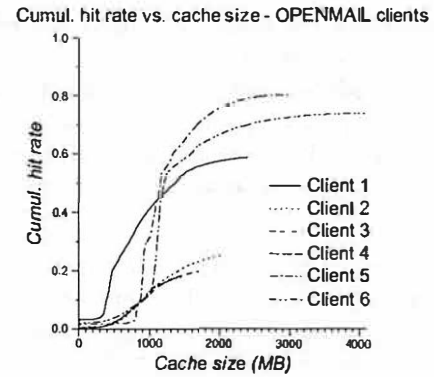
**Table 9:** Per-client mean latencies (in ms) for multi-client HTTPD. Latencies are computed using equations 2 and 3 with  $T_a = 0.2$  ms and  $T_d = 5$  ms. Speedups over NONE-LRU, and the geometric mean of all client speedups, are also shown.

than demoted blocks, it should not discard them immediately.

Given this analysis, we expected DEMOTÉ to post less impressive results than adaptive schemes, and indeed it did, as shown in table 9: a 0.55x slowdown in mean latency over NONE-LRU. On the other hand, DEMOTÉ-ADAPT-EXP achieved a 1.18x speedup. DEMOTÉ-ADAPT-UNI achieved a 0.91x slowdown, which we attribute to demoted blocks being much more valuable than disk-read ones, but the cache with uniform segments devoting too little of its space to them compared to the one with exponential segments.

#### 5.4 The OPENMAIL workload

The OPENMAIL workload comes from a trace of a production e-mail system running the HP OpenMail application for 25,700 users, 9,800 of whom were active during the hour-long trace. The system consisted of six



**Figure 11:** Cumulative hit rate vs. cache size for OPENMAIL.

Client	1	2	3	4	5	6
NONE-LRU						
Mean lat.	2.96	4.52	4.54	4.47	1.79	1.78
DEMOTÉ (mean speedup 1.15x)						
Mean lat.	2.32	4.08	4.27	4.35	1.27	1.67
Speedup	1.28x	1.11x	1.06x	1.03x	1.41x	1.07x
DEMOTÉ-ADAPT-UNI (mean speedup 1.07x)						
Mean lat.	2.66	4.34	4.42	4.46	1.44	1.79
Speedup	1.11x	1.04x	1.03x	1.00x	1.24x	0.99x
DEMOTÉ-ADAPT-EXP (mean slowdown 0.88x)						
Mean lat.	3.03	4.60	4.60	4.54	2.53	2.47
Speedup	0.97x	0.98x	0.99x	0.98x	0.71x	0.72x

**Table 10:** Per-client mean latencies (in ms) for OPENMAIL. Latencies are computed using equations 2 and 3 with  $T_a = 0.2$  ms and  $T_d = 5$  ms. Speedups over NONE-LRU, and the geometric mean of all client speedups, are also shown.

HP 9000 K580 servers running HP-UX 10.20, each with 6 CPUs, 2 GB of memory, and 7 SCSI interface cards. The servers were attached to four EMC Symmetrix 3700 disk arrays. At the time of the trace, the servers were experiencing some load imbalances, and one was I/O bound.

Figure 11 suggests that 2 GB client caches would hold the entire working set for all but two clients. To obtain more interesting results, we simulated six clients with 1 GB caches connected to an array with a 6 GB cache.

OPENMAIL is a disjoint workload, and thus should obtain speedups from exclusive caching. If we assume that each client uses a sixth (1 GB) of the array cache, then each client has an aggregate of 2 GB to hold its workload, and we see from figure 11 that an array under exclusive caching array should obtain a significant increase in array cache hit rate, and a corresponding reduction in mean latency.

As with DB2, our results (table 10) bear out our expectations: DEMOTÉ, which aggressively discards read blocks and holds demoted blocks in the array, obtained a 1.15x speedup over NONE-LRU. DEMOTÉ-ADAPT-UNI and DEMOTÉ-ADAPT-EXP fared less well, yielding a 1.07x speedup and 0.88x slowdown respectively.

## 5.5 Summary

The clear benefits from single-client workloads are not so easily repeated in the multi-client case. For largely disjoint workloads, such as DB2 and OPENMAIL, the simple DEMOTE scheme does well, but it falls down when there is a large amount of data sharing. On the other hand, the adaptive demotion schemes do well when simple DEMOTE fails, which suggests that a mechanism to switch between the two may be helpful.

Overall, our results suggests that even when demotion-based schemes seem not to be ideal, it is usually possible to find a setting where performance is improved. In the enterprise environments we target, such tuning is an expected part of bringing a system into production.

## 6 Related work

The literature on caching in storage systems is large and rich, so we only cite a few representative samples. Much of it focuses on predicting the performance of an existing cache hierarchy [6, 24, 35, 34], describing existing I/O systems [17, 25, 39], and determining when to flush write-back data to disk [21, 26, 41]. Real workloads continue to demonstrate that read caching has considerable value in arrays, and that a small amount of non-volatile memory greatly improves write performance [32, 39].

We are not the first to have observed the drawbacks of inclusive caching. Muntz *et al.* [27, 28] show that intermediate-layer caches for file servers perform poorly, and much of the work on cache replacement algorithms is motivated by this observation [21, 24, 30, 48]. Our DEMOTE scheme, with alternative array cache replacement policies, is another such remedy.

Choosing the correct cache replacement policy in an array can improve its performance [19, 21, 30, 35, 48]. Some studies suggest using least-frequently-used [15, 46] or frequency-based [30] replacement policies instead of LRU in file servers. MRU [23] or next-block prediction [29] policies have been shown to provide better performance for sequential loads. LRU or clocking policies [10] can yield acceptable results for database loads; for example, the IBM DB2 database system [36] implements an augmented LRU-style policy.

Our DEMOTE operation can be viewed as a very simple form of a client-controlled caching policy [7], which could be implemented using the “write to cache” operation available on some arrays (e.g., those from IBM [3]). The difference is that we provide no way for the client to control which blocks the array should replace, and we trust the client to be well-behaved.

Recent studies of cooperative World Wide Web caching

protocols [1, 20, 47] look at policies beyond LRU and MRU. Previously, analyses of web request traces [2, 5, 8] showed the file popularity distributions to be Zipf-like [49]. It is possible that schemes tuned for these workloads will perform as well for the sequential or random access patterns found in file system workloads, but a comprehensive evaluation of them is outside the scope of this paper. In addition, web caching, with its potentially millions of clients, is targeted at a very different environment than our work.

Peer-to-peer cooperative caching studies are relevant to our multi-client case. In the “direct client cooperation” model [9], active clients offload excess blocks onto idle peers. No inter-client sharing occurs—cooperation is simply a way to exploit otherwise unused memory. The GMS global memory management project considers finding the nodes with idle memory [13, 42]. Cooperating nodes use approximate knowledge of the global memory state to make caching and ejection decisions that benefit a page-faulting client and the whole cluster.

Perhaps the closest work to ours in spirit is a global memory management protocol developed for database management systems [14]. Here, the database server keeps a directory of pages in the aggregate cache. This directory allows the server to forward a page request from one client to another that has the data, request that a client demote rather than discard the last in-memory copy of a page, and preferentially discard pages that have already been sent to a client. We take a simpler approach: we do not track which client has what block, and thus cannot support inter-client transfers—but we need neither a directory nor major changes to the SCSI protocol. We rely on a high-speed network to perform DEMOTE eagerly (rather than first check to see if it is worthwhile) and we do not require a (potentially large) data structure at the array to keep track of what blocks are where. Lower complexity has a price: we are less able to exploit block sharing between clients.

## 7 Conclusion

We began our study with a simple idea: that a DEMOTE operation might make array caches more exclusive and thus achieve better hit rates. Experiments with simple synthetic workloads support this hypothesis; moreover, the benefits are reasonably resistant to reductions in SAN bandwidth and variations in array cache size. Our hypothesis is further supported by 1.04–2.20× speedups for most single-client real-life workloads we studied—and these are significantly larger than several results for other cache improvement algorithms.

The TPC-H system parameters show why making ar-

ray caches more exclusive is important in large systems: cache memory for the client and arrays represented 32% of the total system cost of \$1.55 million [18]. The ability to take full advantage of such large investments is a significant benefit; reducing their size is another.

Using multiple clients complicates the story, and our results are less clear-cut in such systems. Although we saw up to a  $1.5\times$  speedup with our exclusive caching schemes, we incurred a slowdown with the simple DEMOTE scheme when clients shared significant parts of the working set. Combining adaptive cache-insertion algorithms with demotions yielded improvements for these shared workloads, but penalized disjoint workloads. However, we believe that it would not be hard to develop an automatic technique to switch between these simple and adaptive modes.

In conclusion, we suggest that the DEMOTE scheme is worth consideration by system designers and I/O architects, given our generally positive results. Better yet, as SAN bandwidth and cache sizes increase, its benefits will likely increase, and not be wiped out by a few months of processor, disk, or memory technology progress.

## 8 Acknowledgments

We would like to thank Greg Ganger, Garth Gibson, Richard Golding, and several of our colleagues for their feedback and support, as well as all the authors of Pantheon. We would also like to thank Liddy Shriver, our USENIX shepherd, for her feedback and help.

## References

- [1] M. F. Arlitt, L. Cherkasova, J. Dillcy, R. Friedrich, and T. Jin. Evaluating content management techniques for web proxy caches. *Performance Evaluation Review*, 27(4):3–11, Mar. 2000.
- [2] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *Proc. of SIGMETRICS 1996*, pages 126–137. July 1996.
- [3] E. Bachmat, EMC. Private communication, Apr. 2002.
- [4] D. Black, EMC. Private communication, Feb. 2002.
- [5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. of the 18th Ann. Joint Conf. of the IEEE Computer and Communications Soc.*, volume 1–3, Mar. 1999.
- [6] D. Buck and M. Singha. An analytic study of caching in computer-systems. *Journal of Parallel and Distributed Computing*, 32(2):205–214, Feb. 1996. Erratum published in 34(2):233, May 1996.
- [7] P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. In *Proc. of the USENIX Assoc. Summer Conf.*, pages 171–182. June 1994.
- [8] M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic evidence and possible causes. In *Proc. of SIGMETRICS 1996*, pages 160–169. July 1996.
- [9] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, pages 267–280. Nov. 1994.
- [10] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Trans. on Database Systems*, 9(4):560–595, Dec. 1984.
- [11] EMC Corporation. Symmetrix 3000 and 5000 enterprise storage systems product description guide. <http://www.emc.com/products/product.pdfs/pdg/symm.3.5.pdg.pdf>, Feb. 1999.
- [12] EMC Corporation. Symmetrix 8000 enterprise storage systems product description guide, Mar. 2001.
- [13] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing global memory management in a workstation cluster. In *Proc. of the 15th Symp. on Operating Systems Principles*, pages 201–212. Dec. 1995.
- [14] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server DBMS architectures. In *Proc. of the 18th Very Large Database Conf.*, pages 596–609. Aug. 1992.
- [15] K. Froese and R. B. Bunt. The effect of client caching on file server workloads. In *Proc. of the 29th Hawaii International Conference on System Sciences*, pages 150–159, Jan. 1996.
- [16] G. R. Ganger and Y. N. Patt. Using system-level models to evaluate I/O subsystem designs. *IEEE Trans. on Computers*, 47(6):667–678, June 1998.
- [17] C. P. Grossman. Evolution of the DASD storage control. *IBM Systems Journal*, 28(2):196–226, 1989.
- [18] Hewlett-Packard Company. HP 9000 N4000 Enterprise Server using HP-UX 11.00 64-bit and Informix Extended Parallel Server 8.30FC2: TPC-H full disclosure report, May 2000.
- [19] S. Jiang and X. Zhuang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. of SIGMETRICS 2002*. June 2002.
- [20] S. Jin and A. Bestavros. Popularity-aware greedy-dual-size web proxy caching algorithms. In *Proc. of the 20th Intl. Conf. on Distributed Computing Systems*, pages 254–261. Apr. 2000.
- [21] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk performance. *IEEE Computer*, 27(3):38–46, Mar. 1994.
- [22] E. D. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27(2):155–164, Nov. 1994.
- [23] K. Korner. Intelligent caching for remote file service. In *Proc. of the 10th Intl. Conf. on Distributed Computing Systems*, pages 220–226. May 1990.
- [24] B. McNutt. I/O subsystem configurations for ESA: New roles for processor storage. *IBM Systems Journal*, 32(2):252–264, 1993.
- [25] J. Menon and M. Hartung. The IBM 3990 disk cache. In *Proc. of COMPCON 1988, the 33rd IEEE Intl. Computer Conf.*, pages 146–151, June 1988.
- [26] D. W. Miller and D. T. Harper. Performance analysis of disk cache write policies. *Microprocessors and Microsystems*, 19(3):121–130, Apr. 1995.
- [27] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems — or — your cache ain't nuthin' but trash. In *Proc. of the USENIX Assoc. Winter Conf.* Jan. 1992.
- [28] D. Muntz, P. Honeyman, and C. J. Antonelli. Evaluating delayed write in a multilevel caching file system. In *Proc. of IFIP/IEEE Intl. Conf. on Distributed Platforms*, pages 415–429. Feb.–Mar. 1996.

- [29] E. Rahm and D. F. Ferguson. Cache management algorithms for sequential data access. Research Report RC15486, IBM T.J. Watson Research Laboratories, Yorktown Heights, NY, 1993.
- [30] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. of SIGMETRICS 1990*, pages 132–142. May 1990.
- [31] C. Ruemmler and J. Wilkes. A trace-driven analysis of disk working set sizes. Tech. Rep. HPL-OSR-93-23, HP Laboratories, Palo Alto, CA, Apr. 1993.
- [32] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *Proc. of the USENIX Assoc. Winter Conf.*, pages 405–420. Jan. 1993.
- [33] C. Ruemmler and J. Wilkes. An introduction to disk drive modelling. *IEEE Computer*, 27(3):17–28, Mar. 1994.
- [34] A. J. Smith. Bibliography on file and I/O system optimization and related topics. *Operating Systems Review*, 15(4):39–54, Oct. 1981.
- [35] A. J. Smith. Disk cache-miss ratio analysis and design considerations. *ACM Trans. on Computer Systems*, 3(3):161–203, Aug. 1985.
- [36] J. Z. Teng and R. A. Gumaer. Managing IBM Database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–218, 1984.
- [37] Transaction Processing Performance Council. TPC benchmark H, Standard Specification Revision 1.3.0. <http://www.tpc.org/tpch/spec/h130.pdf>, June 1999.
- [38] Transaction Processing Performance Council. TPC benchmark C, Standard Specification Version 5. [http://www.tpc.org/tpcc/spec/tpcc\\_current.pdf](http://www.tpc.org/tpcc/spec/tpcc_current.pdf), Feb. 2001.
- [39] K. Treiber and J. Menon. Simulation study of cached RAID5 designs. In *Proc. of the 1st Conf. on High-Performance Computer Architecture*, pages 186–197. Jan. 1995.
- [40] M. Uysal, A. Acharya, and J. Saltz. Requirements of I/O systems for parallel machines: An application-driven study. Tech. Rep. CS-TR-3802, Dept. of Computer Science, University of Maryland, College Park, MD, May 1997.
- [41] A. Varma and Q. Jacobson. Destage algorithms for disk arrays with nonvolatile caches. In *Proc. of the 22nd Ann. Intl. Symp. on Computer Architecture*, pages 83–95. June 1995.
- [42] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, and A. R. Karlin. Implementing cooperative prefetching and caching in a globally managed memory system. In *Proc. of SIGMETRICS 1998*, pages 33–43. June 1998.
- [43] D. Voigt. HP AutoRAID field performance. HP World talk 3354, <http://www.hpl.hp.com/SSP/papers/>, Aug. 1998.
- [44] J. Wilkes. The Pantheon storage-system simulator. Tech. Rep. HPL-SSP-95-14 rev. 1, HP Laboratories, Palo Alto, CA, May 1996.
- [45] J. Wilkes, R. Golding, C. Smelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Trans. on Computer Systems*, 14(1):108–136, Feb. 1996.
- [46] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network file servers. In *Proc. of the 13th Intl. Conf. on Distributed Computing Systems*, pages 2–11. May 1993.
- [47] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. The scale and performance of cooperative web proxy caching. In *Proc. of the 17th Symp. on Operating Systems Principles*, pages 16–31. Dec. 1999.
- [48] Y. Zhou and J. F. Philbin. The Multi-Queue replacement algorithm for second level buffer caches. In *Proc. of the USENIX Ann. Technical Conf.*, pages 91–104. June 2001.
- [49] G. K. Zipf. *Human Behavior and Principle of Least Effort*. Addison-Wesley Press, Cambridge, MA, 1949.





# Bridging the Information Gap in Storage Protocol Stacks

Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau  
*Department of Computer Sciences, University of Wisconsin, Madison*  
{tedenehy, dusseau, remzi}@cs.wisc.edu

## Abstract

*The functionality and performance innovations in file systems and storage systems have proceeded largely independently from each other over the past years. The result is an information gap: neither has information about how the other is designed or implemented, which can result in a high cost of maintenance, poor performance, duplication of features, and limitations on functionality. To bridge this gap, we introduce and evaluate a new division of labor between the storage system and the file system. We develop an enhanced storage layer known as Exposed RAID (E×RAID), which reveals information to file systems built above; specifically, E×RAID exports the parallelism and failure-isolation boundaries of the storage layer, and tracks performance and failure characteristics on a fine-grained basis. To take advantage of the information made available by E×RAID, we develop an Informed Log-Structured File System (I-LFS). I-LFS is an extension of the standard log-structured file system (LFS) that has been altered to take advantage of the performance and failure information exposed by E×RAID. Experiments reveal that our prototype implementation yields benefits in the management, flexibility, reliability, and performance of the storage system, with only a small increase in file system complexity. For example, I-LFS/E×RAID can incorporate new disks into the system on-the-fly, dynamically balance workloads across the disks of the system, allow for user control of file replication, and delay replication of files for increased performance. Much of this functionality would be difficult or impossible to implement with the traditional division of labor between file systems and storage.*

## 1 Introduction

A chasm exists in the world of file storage and management. Though a hierarchical file system of directories and byte-accessible files has been the norm for almost 30 years [27], the internals of file systems and underlying storage systems have evolved substantially, improving both performance [23] and functionality [33].

In file systems, many approaches have been developed to improve performance, including read-optimized inode and file placement [23], logging of writes [30], improved meta-data update methods [39], more scalable internal

data structures [41], and off-line reorganization strategies [22]. However, almost all such techniques have been developed under the assumption that the file system will be run upon a single, traditional disk.

More recently, storage systems have also received much attention. For example, “smart” disks can improve read or write performance with block remapping techniques [11, 13, 49]. For I/O-intensive workloads, multiple-disk storage systems have been well studied in the research community [26, 51], and have achieved success in the storage industry.

These high-end storage systems provide the illusion of a single, fast disk to unsuspecting file systems above, but internally manage both parallelism and redundancy to optimize for performance, capacity, or even both [51]. Analogous to file systems, storage systems are often developed with a single (FFS-like) file system in mind.

While these changes in both file systems and parallel disk systems have been substantial, they have also been separate, and the result is an *information gap*: the file system does not understand the true nature of the storage system it runs upon, and the storage system cannot comprehend the semantic relations between the blocks it stores. In addition, each is unaware of the state the other tracks and the optimizations that the other performs.

This gap arose from a historical source: the hardware/software boundary. File systems have traditionally expected a block-based read/write interface to storage, because that interface is quite similar to what a single disk exports. With the advent of hardware-based RAID systems [26], storage vendors took advantage of the freedom to innovate behind this interface, and thus developed high-performance, high-capacity systems that appeared as a single, large, and fast disk to the file system. No software modifications were required of the host operating system, and file systems continued to operate correctly, in spite of the fact that they were often optimized for a single-disk system. In this case, ignorance was bliss; the arrangement was simple and worked well.

However, the boundary between file system and storage system is changing, migrating towards a software-structuring technique rather than an interface necessitated by hardware. Software RAID drivers are available on many platforms [7], and with the advent of network-

attached storage [14], client-side striping software can replace the need for hardware-based RAID systems entirely. Such software-based RAID systems are particularly attractive due to their low cost, e.g., in a Linux-based system, one incurs only the cost of the machine and disks.

We term the arrangement of a file system layer on top of a software storage layer a “storage protocol stack,” akin to networking protocol stacks that are prominent in communication networks [8]. There are some similarities between the two: layering is known to simplify system design, though potentially at the cost of performance [47]. However, a crucial difference exists: the layers that comprise network protocol stacks are derived by design, with the architects carefully deciding where each specific element should be placed. The storage protocol stack, however, has not been developed in a single, coherent manner; the end result is not only poor performance but also the potential for duplication in implementation and limitations on functionality.

For example, performance may suffer if the model that the file system has of the storage layer is not accurate; thus, layout optimizations that work well on a single, traditional disk may not be appropriate when the logical-block to physical-block mapping is unknown [51]. Feature duplication is also a potential pitfall. For example, a log-structured file system [30] could be layered on top of a disk array that performs logging [40, 51], duplicating work and increasing system complexity unnecessarily. Finally, functionality may be limited, as certain pieces of information only live at one layer of the system. For example, the storage system does not know what blocks constitute a file and thus cannot perform per-file operations, and it does not know that a block is no longer live after a file deletion, and thus cannot optimize the system in ways possible had that knowledge been available.

Thus, we believe that the time is ripe to re-examine the division of labor between the file system and storage system layers, in an attempt to understand the best way to structure the storage protocol stack. Specifically, for each piece of storage functionality, we wish to understand where it is most easily and effectively implemented. We believe the problem is particularly germane at this time, with the move towards network-attached storage (and their proposed higher-level disk interfaces) under way [14].

In this paper, we take a first step towards our goal by exploring a single point in the spectrum of possible designs. To bridge the file system/storage system information gap, we develop and evaluate a new division of labor between the file system and storage. In this realignment, the storage layer exposes parallelism and failure isolation boundaries in part or full to file systems built above, and provides on-line performance and fail-

ure characteristics. We call this layer the *Exposed RAID* layer (E×RAID).

To take advantage of the information provided by E×RAID, we introduce an *Informed LFS* (I-LFS), an enhancement of a log-structured file system [30, 37]. By combining the performance and failure information presented by E×RAID along with file-system specific knowledge, I-LFS is more flexible and manageable than a traditional file system, and can deliver higher performance and availability as well. For example, adding a disk to I-LFS on-line is easily accomplished; further, I-LFS accounts for the potential heterogeneity introduced by a new disk, and dynamically balances load across the disks of the system, whatever their rates. I-LFS also increases the flexibility of storage by enabling user control over redundancy on a per-file basis, and implements lazy mirroring to defer replication to a later time, potentially increasing performance of the system at a slight decrease in reliability. Crucial to I-LFS/E×RAID is the implementation of the aforementioned benefits without a significant increase in overall complexity (and thus maintainability) of the storage protocol stack. Via careful design, all the functionality mentioned above is implemented with only a 19% increase in overall code size as compared to a traditional system.

However, I-LFS/E×RAID is not a panacea. In particular, we find that managing redundancy within the file system can be somewhat onerous, requiring the careful placement of inodes and data blocks to ensure efficient operation under failure. Further, extending the traditional file system structure to support the enhanced functionality of I-LFS was sometimes an arduous task; perhaps a redesign of the age-old vnode layer to support informed file systems is warranted.

The rest of the paper is structured as follows. We begin with a discussion of related work in Section 2. In Section 3, we give an overview of our approach, and then we describe E×RAID and I-LFS in Sections 4 and 5, respectively. Then, in Section 6, we present an evaluation of our system. We present a discussion in Section 7, future work in Section 8, and conclude in Section 9.

## 2 Related Work

Part of our motivation for “informing” the file system of the nature of the storage system is reminiscent of work on the Berkeley Fast File System (FFS) [23]. FFS is an early demonstration of the benefits of having a low-level understanding of disk technology; by colocating correlated inodes and data blocks, performance was improved, especially as compared to the old Unix file system. Our work has the same goal, but with multi-disk storage systems in mind; however, we believe that

the file system should base its decisions upon reliably-obtained information about the characteristics of storage, instead of relying upon assumptions which may or may not hold across time (e.g., that seek costs dominate rotational costs).

Roselli *et al.* discuss the file system/storage system gap in their talk on file system fingerprinting [29]. Their solution is to enrich the interface between file systems and storage systems, by giving the storage system more information about which blocks are related, and which blocks are likely to be accessed again in the near future. Thus, their approach gives the storage system some of the information that the file system might have collected, and presumes that the storage layer can make good use of such information. One potential problem with such an approach is that it may require agreement on a particular set of interfaces among cooperating storage vendors and file-system implementors.

Another example of the benefits of low-level knowledge of disk characteristics is found in Schindler *et al.*'s recent work on track-aligned extents [36]. Therein, the authors explore the range of performance improvements possible when allocating and accessing data on disk-track boundaries, thereby avoiding rotational latency and track-crossing overheads in a single-disk setting. In contrast, E×RAID exposes disk boundaries of a RAID to file systems above, and not such detailed lower-level information; in the future, it would be interesting to investigate the benefits of having lower-level knowledge of the specifics of a RAID-based storage system.

Network Appliance pioneered some of the ideas we discuss here in their work on file server appliances [16]. In the development of WAFL, a write-anywhere file layout technique, Hitz *et al.* hint at how some information normally hidden inside of the RAID layer can be taken advantage of by a file system. For example, they ensure that writes to the RAID-4 layer occur in full-stripe-sized units, and thus avoid the small-write penalty that normally manifests itself on RAID-4 and RAID-5 systems. We take this a step further by formalizing the E×RAID layer, showing that a traditional file system can easily be modified to take advantage of the information provided by E×RAID, and demonstrating that a broader range of optimizations are attainable within such a framework.

Volume managers have long been used to ease the management of storage across multiple devices [44]. The E×RAID layer is simply a new type of volume manager that exposes more information to file systems (specifically, on-line performance and failure information); further, E×RAID is built with the presupposition that a single mounted file system will utilize multiple "volumes" for its data, whereas most volume managers assume that there is a one-to-one mapping between each mounted file system and a volume. One volume manager

that is similar to E×RAID is the Pool Driver, a volume manager for SANs that has a "sub-pool" concept which may be used by a file system to group related data [43]. In that work, the GFS file system uses sub-pools to separate journaled meta-data and normal user data.

Exposing each disk of a storage system to the file system is an extension of the arguments made by Engler and Kaashoek [12]. Therein, the authors argue that software abstractions made by operating systems are fundamentally problematic, as they are often too high-level and thus may limit power and functionality. The authors advocate a solution of exposing all hardware features to the user. Missing from this argument for minimalism is the observation that hardware itself often provides abstractions that users (and operating systems) cannot change. Apropos to data storage, the abstraction put forth by RAID systems is a particularly high-level one, which E×RAID breaks by revealing information that is often hidden from the file system.

Some distributed file systems such as Zebra [15] and xFS [1] manage each disk of the system individually, in a manner similar to I-LFS. However, both of these systems use traditional storage management techniques (such as RAID-5 striping) and do not take advantage of the many potential possibilities that the E×RAID layer makes available. In the future, we hope to extend some of our ideas into the distributed arena, and thus allow for a more direct comparison.

More recently, the NASD object interface has been introduced as a higher-level data repository for SAN-based distributed file systems [14]. This interface allows more advanced functionality to be placed into the storage layer, whereas E×RAID is designed to allow more functionality to be placed within the file system. Earlier work at HP on DataMesh also proposes more sophisticated interfaces for network-attached storage [50].

Our informed approach is also similar to a large body of work in parallel file systems [17, 24]. Most parallel file systems expose disk parallelism, but they allow the application itself, and not the file system, to manage it. Better control over redundancy in a parallel file system has also been proposed [9]. In that work, the computation of parity is put under user control, and in doing so, allows the user to avoid the well-known performance penalty of RAID-4 and RAID-5 under small writes.

### 3 Overview

In the next two sections, we present the design and implementation of E×RAID and I-LFS. Our primary goal in designing the system is to exploit the information made available by E×RAID, thus allowing I-LFS to implement functionality that would be difficult or im-

possible to achieve in a more traditional layering. In particular, we aim to increase: (1) the ease of storage management, (2) performance, especially when considering multiple heterogeneous disks, and (3) functionality, so as to meet the demands of a diverse set of applications.

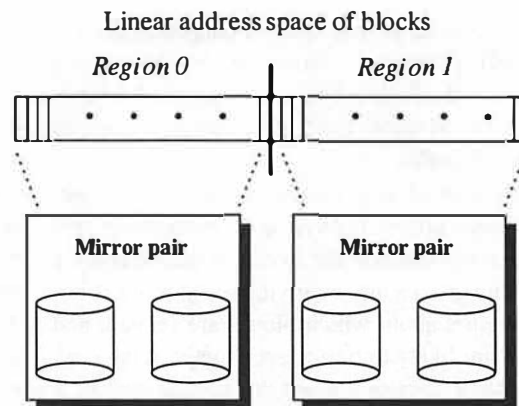
Our primary goal in implementing E×RAID is to facilitate the use of the information provided by E×RAID in the simplest possible way, and to allow non-informed legacy file systems to be built on top of E×RAID with no changes. Our primary goal in implementing I-LFS is to minimize the impact of transforming the file system to utilize the new storage interface. For example, changes that would require a re-design of the vnode layer were ruled out, as that would mandate that all other file systems be changed in order to function in our system. Thus, throughout our implementation effort, we integrate changes into I-LFS in a highly localized and modular fashion – the fewer lines of code that changed, the better.

One question that must be addressed is our decision to modify LFS and not a more traditional (or perhaps more popular) FFS-like or journaling file system. One reason we chose LFS is its natural flexibility in data placement; LFS is a modern example of a “write anywhere” storage system [16, 19]. Write-anywhere systems provide an extra level of indirection such that writes can be placed in any location on the storage medium, and we exploit this aspect of LFS in part of our implementation. However, with this in mind, we do believe that a number of our implementation techniques are general and could be applied to other file systems, and hope to investigate doing so in the future. Those interested in general LFS file system performance issues should consult the work of Rosenblum and Ousterhout [30], or subsequent research by Seltzer *et al.* [37, 38].

All of our software was developed within the context of the NetBSD 1.5 operating system. E×RAID was implemented as a set of hooks on the lower-level block-driver calls, and is described in more detail in Section 4. I-LFS was implemented by extending the NetBSD version of LFS, which is based on the original LFS for BSD Unix [37], and is described in detail in Section 5. We chose the NetBSD version of LFS as it is known to be a relatively stable and solid implementation.

## 4 E×RAID

We now describe the E×RAID storage interface. It consists of two major components: a segmented address space which exposes some or all of the parallelism of the storage system to the file system, and functions used to inform the file system of the dynamic state of the storage system.



**Figure 1: An Example E×RAID Configuration.** The diagram depicts an example E×RAID configuration in which each of two disks is combined into a mirrored pair. Two regions, each half of the size of the total address space, are presented to the client file system. Within a region, the layout performed by the mirror is hidden from the file system.

### 4.1 A Segmented Address Space

A traditional RAID array presents the storage subsystem to the file system as a linear array of blocks, underneath of which the true complexity of the particular RAID scheme is hidden. File systems interact with RAID systems by either reading or writing the blocks. In keeping with our desire to minimize change and preserve backwards compatibility, E×RAID also provides a linear array of blocks which can be read or written as the basic interface.

However, because we wish to expose information about the storage system to the file system, the address space is *segmented*; specifically, it is organized as a series of contiguous *regions*, each of which is mapped directly to a single disk (or set of disks), and these region boundaries are made known to the file system above, if it so desires. For example, in a four-disk storage system with each disk capable of storing  $N$  blocks, the address space E×RAID presents might be segmented as follows: blocks 0 through  $N - 1$  map to disk 0, blocks  $N$  through  $2N - 1$  map to disk 1, and so forth.

By exposing this information, E×RAID enables the file system to understand the performance and failure boundaries of the storage system. As we shall see in later sections, the file system can take advantage of this to place data on a particular region more intelligently, potentially improving performance, reliability, or other aspects of the storage system.

Within E×RAID, a region may represent more than just a single disk. For example, a region could be configured to represent a mirrored pair of disks, or even a RAID-5 collection. Thus, each region can be viewed as a configurable software-based RAID, and the entire

E×RAID address space as a single representation of the conglomeration of such RAID subsystems. In such a scenario, some information is hidden from the file system, but cross-region optimizations are still possible, if more than one region exists. An example of an E×RAID configuration over mirrored pairs is shown in Figure 1.

Allowing each region to represent more than just a single disk has two primary benefits. First, if each region is configured as a RAID (such as a mirrored pair of disks), the file system is not forced to manage redundancy itself, though it can choose to do so if so desired. Second, this arrangement allows for backwards compatibility, as E×RAID can be configured as a single striped, mirrored, or RAID-5 region, thus allowing unmodified file systems to use it without change.

## 4.2 Dynamic Information

Although the segmented address space exposes the nature of the underlying disk system to the file system (either in part or in full), this knowledge is often not enough to make intelligent decisions about data placement or replication. Thus, the E×RAID layer exposes dynamic information about the state of each region to the file system above, and it is in this way that E×RAID distinguishes itself from traditional volume managers.

Two pieces of information are needed. First, the file system may desire to have *performance* information on a per-region basis. The E×RAID layer tracks queue lengths and current throughput levels, and makes these pieces of information available to the file system. Historical tracking of information is left to the file system.

Second, the file system may wish to know about the resilience of each region, *i.e.*, when failures occur, and how many more failures a region can tolerate. Thus, E×RAID also presents this information to the file system. For example, in Figure 1, the file system would know that each mirror pair could tolerate a single disk failure, and would be informed when such a failure occurs. The file system could then take action, perhaps by directing subsequent writes to other regions, or even by moving important data from the “bad” region into other, more reliable portions of the E×RAID address space.

## 4.3 Implementation

In our current implementation, E×RAID is implemented as a thin layer between the file system and the storage system. In order to implement a striped, mirrored, or RAID-5 region, we simply utilize the standard software RAID layer provided with NetBSD. However, our prototype E×RAID layer is not completely generalized as of this date, and thus in its current form would require some effort to allow a file system other than I-LFS to utilize it.

The segmented address space is built by interposing on the *vnode strategy* call, which allows us to remap requests from their logical block number within the virtual address space presented by E×RAID into a physical disk number and block offset, which can then be issued to underlying disk or RAID.

Dynamic performance information is collected by monitoring the current performance levels of reads and writes. In the prototype, region boundaries, failure information, and performance levels (throughput and queue length) are tracked in the low-levels of the file system. A more complete implementation would make the information available through an *ioctl()* interface to the E×RAID device. Also note that we focus primarily on utilizing the performance information in this paper.

## 5 I-LFS

We now describe the I-LFS file system. Our current design has four major pieces of additional functionality, as compared to the standard LFS: on-line expandability of the storage system, dynamic parallelism to account for performance heterogeneity, flexible user-managed redundancy, and lazy mirroring of writes. In sum total, these added features make the system more manageable (the administrator can easily add a new disk, without worry of configuration), more flexible (users have control over if replication occurs), and have higher performance (I-LFS delivers the full bandwidth of the system even in heterogeneous configurations, and flexible mirroring avoids some of the costs of more rigid redundancy schemes). For most of the discussion, we focus on the case that most separates I-LFS/E×RAID from a traditional RAID, where the E×RAID layer exposes each disk of the storage system as a separate region to I-LFS.

### 5.1 On-Line Expansion and Contraction

**Design:** The ability to upgrade a storage system incrementally is crucial. As the performance or capacity demands of a site increase, an administrator may need to add more disks. Ideally, such an addition should be simple to perform (*e.g.*, a single command issued by the administrator, or an automatic addition when the disk is detected by the hardware), require no down-time (thus keeping availability of storage high), and immediately make the extra performance and capacity of the new disk available.

In older systems, on-line expansion is not possible. Even if the storage system could add a new disk on-the-fly, it is likely the case that an administrator would have to unmount the partition, expand it (perhaps with a tool similar to that described in [46]), and then re-mount the



file system. Worse, some systems require that a new file system be built, forcing the administrator to restore data from tape. More modern volume managers [48] allow for on-line expansion, but still need file system support.

Thus, our I-LFS design includes the ability to incorporate new disks (really, new E×RAID regions) on-line with a single command given to the file system. No complicated support is necessitated across many layers of the system. If the hardware supports hot-plug and detection of new disks without a power-cycle, I-LFS can add new disks without any down time and thus reduction in data availability. Overall, the amount of work an administrator must put forth to expand the system is quite small.

Contraction is also important, as the removal of a region should be as simple as the addition of one. Therefore, we also incorporate the ability to remove a region on the fly. Of course, if the file system has been configured in a non-redundant manner, some data will likely be lost. The difference between I-LFS and a traditional system in this scenario is that I-LFS knows exactly which files are available and can deliver them to applications.

**Implementation:** To allow for on-line expansion and contraction of storage, the file system views regions that have not yet been added as extant and yet fully utilized; thus, when a new region is added to the system, the blocks of that disk are made available for allocation, and the file system will immediately begin to write data to them. Conversely, a region that is removed is viewed as fully allocated. This technique is general and could be applied to other file systems, and similar ideas have been used elsewhere [16].

More specifically, because a log-structured file system is composed of a collection of LFS segments, it is natural to expand capacity within I-LFS by adding more free segments. To implement this functionality, the `newfs.ilfs` program creates an expanded LFS segment table for the file system. The entries in the segment table record the current state of each segment. When a new E×RAID region is added to the file system, the pertinent information is added to the superblock, and an additional portion of the segment table is activated. This approach limits the number of regions that can be added to a fixed number (currently, 16); for more flexible growth, the segment table could be placed in its own file and expanded as necessary.

## 5.2 Dynamic Parallelism

**Design:** One problem introduced by the flexibility an administrator has in growing a system is the increased potential for performance heterogeneity in the disk subsystem; in particular, a new disk or E×RAID segment may have different performance characteristics than the other disks of the system. In such a case, traditional

striping and RAID schemes do not work well, as they all assume that disks run at identical rates [4, 10].

Traditionally, the presence of multiple disks is hidden by the storage layer from the file system. Thus, current systems must handle any disk performance heterogeneity in the storage layer – the file system does not have enough information to do so itself. The research community has proposed schemes to deal with static disk heterogeneity [3, 10, 32, 52], though many of these solutions require careful tuning by an administrator. As Van Jacobsen notes, “Experience shows that anything that needs to be configured will be misconfigured” [18].

Further complicating the issue is that the delivered performance of a device could change over time. Such changes could result from workload imbalances, or perhaps from the “fail-stutter” nature of modern devices, which may present correct operation but degraded performance to clients [5]. Even if more advanced heterogeneous data layout schemes are utilized, they will not work well under dynamic shifts in performance.

To handle such static and dynamic performance differences among disks, we include a dynamic segment placement mechanism within I-LFS [4]. A segment can logically be written to any free space in the file system; we exploit this by writing segments to E×RAID regions in proportion to their current rate of performance, exploiting the dynamic state presented to the file system by E×RAID. By doing so, we can dynamically balance the write load of the system to account for static or dynamic heterogeneity in the disk subsystem. Note that if performance of the disks is roughly equivalent, this dynamic scheme will degenerate to standard RAID-0 striping of segments across disks.

This style of dynamic placement could also be performed in a more traditional storage system (*e.g.*, AutoRAID has the basic mechanisms in place to do so [51]). However, doing so unduly adds complexity into the system, as *both* the file system and the storage system have to track where blocks are placed; by pushing dynamic segment placement into the file system, overall complexity is reduced, as the file system already tracks where the blocks of a file are located.

**Implementation:** The original version of LFS allocates segments sequentially based on availability; in other words, all free segments are treated equally. To better manage parallelism among disks in I-LFS, we develop a *segment indirection* technique. Specifically, we modify the `ilfs.newseg()` routine to invoke a data placement strategy. The `ilfs.newseg()` routine is used to find the next free segment to write to; here, we alter it to be “region aware”, and thus allow for a more informed segment-placement decision. By choosing disks in accordance with their performance levels (information provided by E×RAID), the load across a set of

heterogeneously-performing regions can be balanced.

The major advantage of our decision to implement this functionality within the `ilfs.newseg()` routine is that it localizes the knowledge of multiple disks to a very small portion of the file system; the vast majority of code in the file system is not aware of the region boundaries within the disk address space, and thus remains unchanged. The slight drawback is that the decision of which region to place a segment upon is made early, before the segment has been written to; if the performance level of the disk changes as the segment fills in a significant way, the placement decision could potentially be a poor one. In practice, we have not found this to be a performance problem.

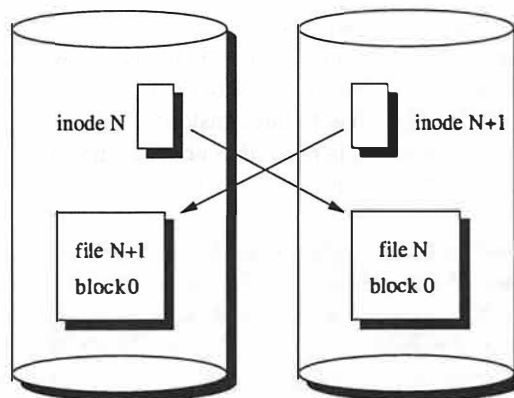
### 5.3 Flexible Redundancy

**Design:** Typically, redundancy is implemented in a one-size-fits-all manner, as a single RAID scheme (or two, as in AutoRAID) is applied to all the blocks of the storage system. The file system is typically neither involved nor aware of the details of data replication within the storage layer. This traditional approach is limiting, as much semantic information is available in the file system as well as in smart users or applications, which could be exploited to improve performance or better utilize capacity.

Thus, in I-LFS, we explore the management of redundancy strictly within the file system, as managing redundancy in the file system provides greater flexibility and control to users. In our current design, we allow users or applications to select whether a file should be made redundant (in particular, if it should be mirrored). If a file is mirrored, users pay the cost in terms of performance and capacity. If a file is not mirrored, performance increases during writes to that file, and capacity is saved, but the chances of losing the file are increased. Turning off redundancy is thus well-suited for temporary files, files that can easily be regenerated, or swap files.

Because I-LFS performs the replication, better accounting is also possible, as the system knows exactly which files (and hence which users) are using which physical blocks. In contrast, with a traditional file system mounted on top of an advanced storage system such as AutoRAID [51], users are charged based on the logical capacity they are using, whereas the true usage of storage depends on access patterns and usage frequency.

Because redundancy schemes are usually implemented within the RAID storage system (where no notion of a file exists), our scheme would not easily be implemented in a traditionally-layered system. The storage system is wholly unaware of which blocks constitute a file and therefore cannot receive input from a user as to which blocks to replicate; only if both the file system



**Figure 2: The “Crossed Pointer” Problem.** *The figure illustrates the problem with using a separate file as a means for redundancy; specifically, even though each element of a file (inode, data block) has been replicated, a single lost disk could still make it difficult to find a particular data block, due to the extra requirement that for each block, a pointer chain to the block must still be live. In the example, the file with inode number  $N$  and its mirror, inode  $N + 1$ , consist of a single data block (block 0). If either disk crashes, it is not possible to find the corresponding data block, even though a copy of it exists on the remaining working disk.*

and storage system were altered could such functionality be realized. In the future, it would be interesting to investigate a range of policies on top of our redundancy mechanisms that automatically apply different redundancy strategies according to the class of a file, akin to how the Elephant file system segregates files for different versioning techniques [33].

**Implementation:** To accomplish our goal of per-file redundancy, we decided to utilize separate and unique meta-data for original and redundant files. This approach is natural within the file system as it does not require changes to on-disk data structures.

In our implementation, we use a straight-forward scheme that assigns even inode numbers to original files and odd inode numbers to their redundant copies. This method has several advantages. Because the original and redundant files have unique inodes, the data blocks can be distributed arbitrarily across disks (given certain constraints described below), thus allowing us to use redundancy in combination with our other file system features. Also, the number of LFS inodes is unlimited because they are written to the log, and the inode map is stored in a regular file which is expanded as necessary. The prime disadvantage of our approach is that it limits redundancy to one copy, but this could easily be extended to an  $N$ -way mirroring scheme by reserving  $N$  i-numbers per file.

One problem introduced by our decision to utilize separate inodes to track the primary and mirrored copy of a file is what we refer to as the “crossed pointer” problem. Figure 2 illustrates the difficulty that can arise.

Simply requiring each component of a file (e.g., the inode, indirect blocks, and data blocks) be replicated is not sufficient to guarantee that all data can be recovered easily under a single disk failure. Instead, we must ensure that each data block is *reachable* under a disk failure; a block being reachable implies that a pointer chain to it exists.

Consider the example in the figure: a file with inode number  $N$  is replicated within inode number  $N + 1$ . Inode  $N$  is located on the first disk, as is the first data block of the mirror copy (file  $N + 1$ ). Inode  $N + 1$  is on the other disk, as is the first data block of the primary copy (file  $N$ ). However, if either disk fails, the first data block is not easily recovered, as the inode on the surviving disk points to the data block on the failed disk. In some file systems, this would be a fatal flaw, as the data block would be unrecoverable. In LFS, it is only a performance issue, as the extra information found within segment summary blocks allows for full recovery; however, a disk crash would mandate a full scan of the disk to recover all data blocks.

There are a number of possible remedies to the problem. For example, one could perform an explicit replication of each inode and all other pointer-carrying structures, such as indirect blocks, doubly-indirect blocks, and so forth. However, this would require the on-disk format to change, and would be inefficient in its usage of disk space, as each inode and indirect block would have four logical copies in the file system.

Instead, we take a much simpler approach of *divide and conquer*. The disks of the system are divided into two sets. When writing a redundant file to disk, I-LFS decides which set the primary copy should be placed within; the redundant copy is placed within the other set. Thus, because no pointers cross from either set into the other, we can guarantee that a single failure will cause no harm (in fact, we can tolerate any number of failures to disks in that set).

Finally, incorporating redundancy into I-LFS also presents us with a difficult implementation challenge: how should we replicate the data and inodes within the file system, without re-writing every routine that creates or modifies data on disk? We develop and apply *recursive vnode invocation* to ease the task. We embellish most I-LFS vnode operations with a short recursive tail; therein, the routine is invoked recursively (with appropriate arguments) if the routine is currently operating on an even i-number and therefore on the primary copy of the data, and if the file is designated for redundancy by the user. For instance, when a file is created using `ilfs_create()`, a recursive call to `ilfs_create()` is used to create a redundant file. The recursion is broken within the call to perform the identical operation to the redundant file.

## 5.4 Lazy Mirroring

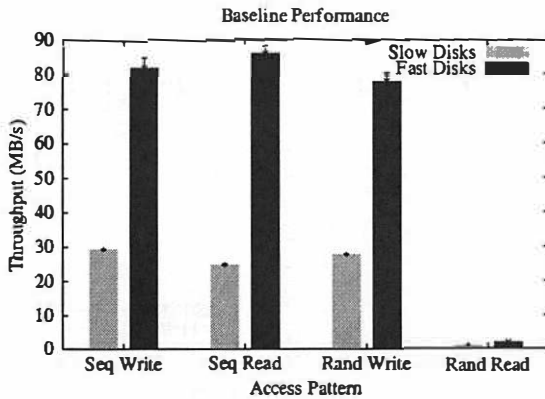
**Design:** User-controlled replication allows users to control *if* replication occurs, but not *when*. As has been shown in previous work, many potential benefits arise in allowing flexible control over when redundant copies are made or parity is updated [9]. Delaying parity updates has been shown to be beneficial in RAID-5 schemes to avoid the small-write problem [34], and could also reduce load under mirrored schemes. Implementing such a feature at the file system level allows the user to decide the window of vulnerability for each file, as losing data in certain files may likely be more tolerable than in others. Note that either of these enhancements would be difficult to implement in a traditional system, as the information required resides in both the file system and RAID, necessitating non-trivial changes to both.

In I-LFS, we incorporate *lazy mirroring* into our user-controlled replication scheme. Thus, users can designate a file as non-replicated, immediately replicated, or lazily replicated. By choosing a lazy replica, the user is willing to increase the chance of data loss for improved performance. Lazy mirroring can improve performance for one of two reasons. First, by delaying file replication, the file system may reduce load under a burst of traffic and defer the work of replication to a later period of lower system load. Second, if a file is written to disk and then deleted before the replication occurs, the cost of replication is removed entirely. As most systems buffer files in memory for a short period of time (e.g., 30 seconds), and file lifetimes have recently been shown to be longer than this on average [28], this second scenario may be more common than previously thought.

**Implementation:** Lazy mirroring is implemented in I-LFS as an embellishment to the file-system cleaner. For files that are designated as lazy replicas, an extra bit is set in the segment usage table indicating their status. When the cleaner scans a segment and finds blocks that need to be replicated, it simply performs the replication directly, making sure to place replicated blocks so as to avoid the “crossed pointer” problem, and associates them with the mirrored inode. When the replication is complete, the bit is cleared. Currently, the file system replicates files after a 2-minute delay, though in the future this could be set directly by the user or application.

## 6 Evaluation

In this section, we present an evaluation of E×RAID and I-LFS. Experiments are performed upon an Intel-based PC with 128 MB of physical memory. The main processor is a 1-GHz Intel Pentium III Xeon, and the system houses four 10,000 RPM Seagate ST318305LC



**Figure 3: Baseline Performance Comparison.** The figure plots the performance of I-LFS/E×RAID under sequential writes, sequential reads, random writes, and random reads. The tests are run on four disks, varying whether the disks used are the four slow disks or the four fast ones. In all cases, requests generated by the tests are 8 KB in size, and the total data-set size is 200 MB.

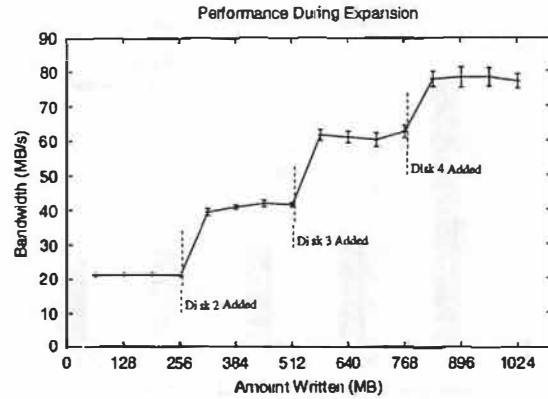
Cheetah 36XL disks (which we will refer to as the “fast” disks), and four 7,200 RPM Seagate ST34572W Barracuda 4XL disks (the “slow” disks). The fast disks can deliver data at roughly 21.6 MB/s each, and the slow disks at approximately 7.5 MB/s apiece. For all experiments, we perform 30 trials and show both the average and standard deviation.

In some experiments, we compare the performance of I-LFS/E×RAID to standard RAID-0 striping. Stripe sizes are chosen so as to maximize performance of the RAID-0 given the workload at hand, making the comparison as fair as possible, or even slightly unfair towards I-LFS/E×RAID.

## 6.1 Baseline Performance

In this first experiment, we demonstrate the baseline performance of I-LFS/E×RAID on top of two different homogeneous storage configurations, one with four slow disks, and one with four fast disks. The experiment consists of sequential write, sequential read, random write, and random read phases (based on patterns generated by the Bonnie [6] and IOzone [25] benchmarks). We perform this experiment to demonstrate that there is no unexpected overhead in our implementation, and that it scales to higher-performance disks effectively.

As we can see in Figure 3, sequential write, sequential read, and random writes all perform excellently, achieving high bandwidth across both disk configurations. Not surprisingly for a log-based file system, random reads perform much more poorly, achieving roughly 0.9 MB/s on the four slow disks, and 1.8 MB/s on the four fast disks, in line with what one would expect from these disks in a typical RAID configuration.



**Figure 4: Storage Expansion.** The graph plots the performance of I-LFS during storage expansion. The experiment begins with I-LFS writing to a single disk. Each time 256 MB is written, a new disk is brought on-line, and I-LFS immediately begins writing to it for increased performance. Disk expansion is accomplished via a simple command, which adds the disk (or region) to the file system without down time.

## 6.2 On-line Expansion

We now demonstrate the performance of the system under writes as disks are added to the system on-line. In this experiment, the disks are already present within the PC, and thus the expansion stresses the software infrastructure and not hardware capabilities.

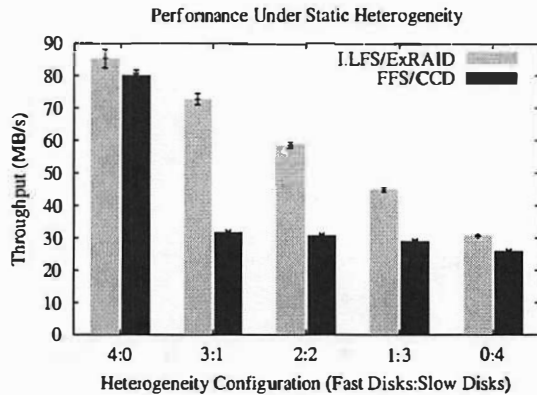
Figure 4 plots the performance of sequential writes over time as disks are added to the system.<sup>1</sup> Along the x-axis, the amount of data written to disk is shown, and the y-axis plots the rate that the most recent 64 MB was committed to disk. As one can see from the graph, I-LFS immediately starts using the disks for write traffic as they are added to the system. However, read traffic will continue to be directed to the original disks for older data. The LFS cleaner could redistribute existing data over the newly-added disks, either explicitly or through cleaning, but we have not yet explored this possibility.

## 6.3 Dynamic Parallelism

We next explore the ability of I-LFS to place segments dynamically in different regions based on the current performance characteristics of the system, in order to demonstrate the ability of I-LFS to react to static and dynamic performance differences across devices.

There are many reasons for performance variation among drives. For example, when new disks are added, they can likely be faster than older ones; further, unexpected dynamic performance variations due to bad-block remapping or “hot spots” in the workload are not uncommon [5], and therefore can also lead to performance

<sup>1</sup>Random writes perform similarly, due to the nature of LFS.



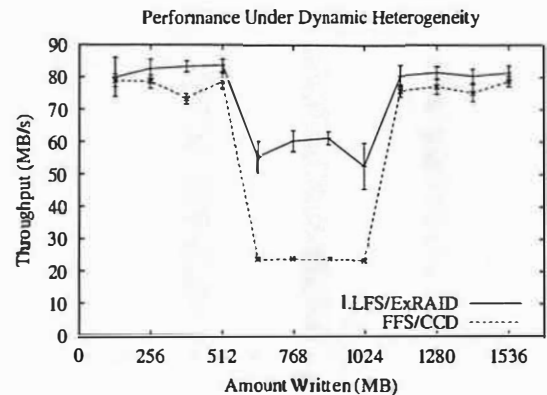
**Figure 5: Static Storage Heterogeneity.** The figure plots the performance of I-LFS versus FFS/CCD with standard RAID-0 striping, both under a series of disk configurations. Along the x-axis, the number of fast and slow disks are varied ( $f:s$  implies  $f$  fast disks and  $s$  slow ones). By adjusting where segments are written dynamically, I-LFS/ExRAID is able to deliver the full bandwidth of disks. In contrast, standard striping performs at the rate of the slowest disk in the system. For each test, 200 MB is written to disk.

heterogeneity across disks. Indeed, the ability to expand the disk system on-line (as shown above) induces a workload imbalance, as read traffic is not directed to the newly-added disks until the cleaner has reorganized data across all of the disks in the system.

We experiment with both static and dynamic performance variations in this subsection. Figure 5 shows the results of our static heterogeneity test. The sequential write performance of I-LFS with its dynamic segment placement scheme is plotted along with FFS on top of the NetBSD concatenated disk driver (CCD) configured to stripe data in a RAID-0 fashion. In all experiments, data is written to four disks. Along the x-axis, we increase the number of slow disks in the system; thus, at the extreme left, all of the four disks are fast ones, at the right they are all slow ones, and in the middle are different heterogeneous configurations.

As we can see in the figure, by writing segments dynamically in proportion to delivered disk performance, I-LFS/ExRAID is able to deliver the full bandwidth of the underlying storage system to applications – overall performance degrades gracefully as more slow disks replace fast ones in the storage system. RAID-0 striping performs at the rate of the slowest disk, and thus performs poorly in any heterogeneous configuration.

We also perform a “misconfiguration” test. In this experiment, we configure the storage system to utilize two partitions on the *same* disk, emulating a misconfiguration by an administrator (similar in spirit to tests performed by Brown and Patterson [7]). Thus, while the disk system appears to contain four separate disks, it really only contains three. In this case, I-LFS/ExRAID



**Figure 6: Dynamic Storage Heterogeneity.** The figure plots the performance of I-LFS/ExRAID and FFS/CCD under a dynamic performance variation. During the experiment, the performance of a single disk is temporarily degraded; the faulty disk delays requests for a fixed time, reducing throughput of the disk from 21.6 MB/s to 5.8 MB/s. By adaptively writing more data to the other disks, I-LFS/ExRAID with dynamic segment placement is better able to adjust to the imbalance and deliver higher throughput.

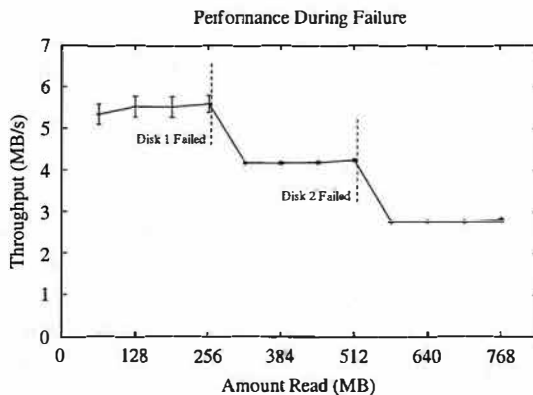
writes data to disk at 65 MB/s, whereas standard striping delivers only 46 MB/s. The dynamic segment striping of I-LFS is successfully able to balance load across the disks, in this case properly assigning less load to each partition within the accidentally over-burdened disk.

In our final heterogeneity experiment, we introduce an artificial “performance fault” into a storage system consisting of four fast disks, in order to confirm that our load balancing works well in the face of dynamic performance variations. Figure 6 shows the performance during a write of both I-LFS/ExRAID with dynamic segment placement and FFS/CCD using RAID-0 striping in a case where a single disk of the four exhibits a performance degradation. After one third of the data is written, a kernel-based utility is used to temporarily delay completed requests from one of the disks. The delay has the effect of reducing its throughput from 21.6 MB/s to 5.8 MB/s. The impaired disk is returned to normal operation after an additional one third of the data is written. As we can see from the figure, I-LFS/ExRAID does a better job of tolerating the fluctuations induced during the second phase of the experiment, improving performance by over a factor of two as compared to FFS/CCD.

## 6.4 Flexible Redundancy

In our first redundancy experiment, we verify the operation of our system in the face of failure. Figure 7 plots the performance of a set of processes performing random reads from redundant files on I-LFS. Initially, the bandwidth of all four disks is utilized by balancing the read load across the mirrored copies of the data. As





**Figure 7: Storage Failure.** The figure plots the random read performance to a set of mirrored files across four disks on I-LFS. At the labeled points in the graph, a disk is taken off-line, and performance decreases because I-LFS can no longer balance the read load between the replicas. Note that in this example, I-LFS/E×RAID can survive any single disk failure; however, after the first failure, I-LFS/E×RAID can only tolerate the loss of the other disk in the set.

the experiment progresses, a disk failure is simulated by disabling reads to one of the disks. I-LFS continues providing data from the available replicas, but overall performance is reduced.

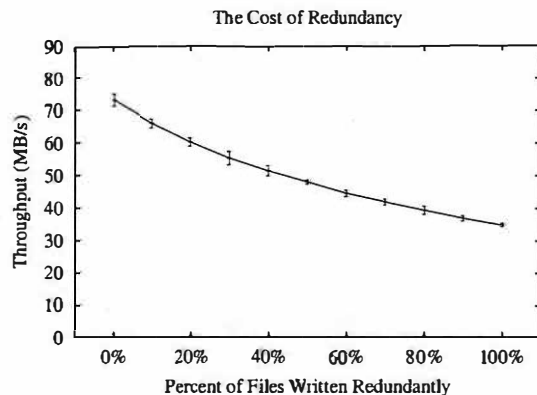
Next, we demonstrate the flexibility of per-file redundancy when the redundancy is managed by the file system. A total of 20 files are written concurrently to a system consisting of four fast disks, while the percentage of those files that are mirrored is increased along the x-axis. The results are shown in Figure 8.

As expected, the net throughput of the system decreases linearly as more files are mirrored, and when all are mirrored, overall throughput is roughly halved. Thus, with per-file redundancy, users “get what they pay for”; if users want a file to be redundant, the performance cost of replication is paid during the write, and if not, the performance of the write reflects the full bandwidth of the underlying disks.

## 6.5 Lazy Mirroring

In our final experiment, we demonstrate some of the performance characteristics of lazy mirroring. Figure 9 plots the write performance to a set of lazily mirrored files. After a delay of 20 seconds, the cleaner begins replicating data, and the normal file system traffic suffers from a small decline in performance. The default replication delay for the system is two minutes in length, but an abbreviated delay is used here to reduce the time of the experiments.

From the figure, we can see the potential benefits of lazy mirroring, as well as its potential costs. If lazily mirrored files are indeed deleted before replication be-



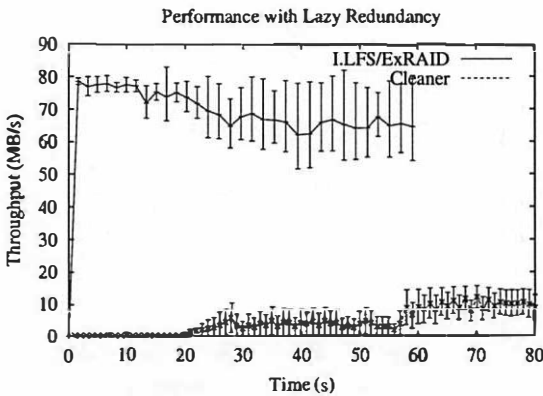
**Figure 8: Per-file Redundancy.** The figure plots the performance of writes to 20 separate files as the percent of those files that are mirrored increases. As more files are mirrored, the net bandwidth of the system drops to roughly half of its peak rate, as expected. The peak bandwidth achieved is lower than the previous experiments due to the increased number of files and subsequent meta-data operations. In each experiment, 200 MB is written out to disk.

gins, the full throughput of the storage layer will be realized. However, if many or all lazily mirrored files are not deleted before replication, the system incurs an extra penalty, as those files must be read back from disk and then replicated, which will affect subsequent file system traffic. Therefore, lazy mirroring should be used carefully, either in systems with highly bursty traffic (*i.e.*, idle time for the lazy replicas to be created), or with files that are easily distinguishable as short-lived.

## 7 Discussion

In implementing I-LFS/E×RAID, we were concerned that by pushing more functionality into the file system, the code would become unmanageably complex. Thus, one of our primary goals is to minimize code complexity. We believe we achieve this goal, integrating the three major pieces of functionality with only an additional 1,500 lines of code, a 19% increase over the original size of the LFS implementation. Of this additional code, roughly half is due to the redundancy management.

From the design standpoint, we find that managing redundancy within the file system has many benefits, but also causes many difficulties. For example, to solve the crossed-pointer problem, we applied a divide-and-conquer technique. By placing the primary copy of a file into one of two sets, and its mirror in the other, we enable fast operation under failure. However, our solution limits data placement flexibility, in that once a file is assigned to a set, any subsequent writes to that file must be written to that set. This limitation affects performance, particularly under heterogeneous configura-



**Figure 9: Lazy Mirroring.** The figure plots the write performance to a set of lazy redundant files on I-LFS with a replication delay of 20 seconds. Peak performance is achieved during the initial portion of the test, but performance is reduced slightly as the cleaner begins replicating data. After the write test completes, the cleaner continues to replicate data in the background.

rations where one set has significantly different performance characteristics than the other. Though we can relax these placement restrictions, *e.g.*, by choosing which disks constitute a set on a per-file basis, the problem is fundamental to our approach to file-system management of redundancy.

From the implementation standpoint, file-system managed redundancy is also problematic, in that the vnode layer is designed with a single underlying disk in mind. Though our recursive invocation technique was successful, it stretched the limits of what was possible in the current framework, and new additions or modifications to the code are not always straightforward to implement. To truly support file-system managed redundancy, a redesign of the vnode layer may be beneficial [31].

## 8 Future Work

A number of possible avenues exist for future research. Most generally, we believe more organizations of the storage protocol stack need to be explored. Which pieces of functionality should be implemented where, and what are the trade-offs? One natural follow-on is to incorporate more lower-level information into ExRAID; the main challenge when exposing new information to the file system is to find useful pieces of information that the file system can readily exploit.

Of course, most file service today spans client and server machines. Thus, we believe it is important to consider how functionality should be split across machines. Which portion of the traditional storage protocol stack should reside on clients, and which portion should reside on the servers? Researchers in distributed file systems

have taken opposing points of view on this, with systems such as Zebra [15] and xFS [1] letting clients do most of the work, whereas the Frangipani/Petal system places most functionality within the storage servers [21, 45].

We also believe cooperative approaches between the file system and storage system may be useful. For example, we found that implementing redundancy in the file system was sometimes vexing; perhaps an approach that shared the responsibility of redundancy across both file system and storage layer would be an improvement. For example, the storage layer could tell the file system which block to use as a mirror of another block, but the file system could decide when to perform the replication.

Even if we decide upon a new storage interface, it may be difficult to convince storage vendors to move away from the tried-and-true standard SCSI interface to storage. Thus, a more pragmatic approach may be to treat the RAID layer as a *gray box*, inferring its characteristics and then exploiting them in the file system, all without modification of the underlying RAID layer [2]. Tools that automatically extract low-level information from disk drives, such as DIXtrac [35] and SKIPPY [42], are first steps towards this goal, with extensions needed to understand the parallel aspects of storage systems.

Finally, we envision many more possible optimizations in our new arrangement of the storage protocol stack. For example, we are currently exploring the notion of *intelligent reconstruction*. The basic idea is simple: if a disk (or region) fails, and I-LFS has duplicated the data upon that disk, I-LFS can begin the reconstruction process itself. The key difference is that I-LFS will only reconstruct live data from that disk, and not the entire disk blindly, as a storage system would, substantially lowering the time to perform the operation. A fringe benefit of intelligent reconstruction is that I-LFS should be able to give preference to certain files over others, reconstructing higher-priority files first and thus increasing the availability of those files under failure.

We also imagine that many optimizations are possible with the LFS cleaner. For example, as data is laid out on disk according to current performance characteristics and access patterns, it may not meet the needs of subsequent potentially non-sequential reads from other applications. Similarly, as new disks are added, the cleaner may want to run in order to lay out older data across the new disks. Thus, the cleaner could be used to reorganize data across drives for better read performance in the presence of heterogeneity and new drives, similar to the work of Neefe *et al.*, but generalized to operate in a heterogeneous multi-disk setting [22].

## 9 Conclusions

In terms of abstractions, block-level storage systems such as SCSI have been quite successful: disks hide low-level details from file systems such as the exact mechanics of arm movement and head positioning, but still export a simple performance model upon which file systems could optimize. As Lampson said: “[...] an interface can combine simplicity, flexibility, and high performance together by solving one problem and leaving the rest to the client” [20]. In early single-disk systems, this balance was struck nearly perfectly.

As storage systems evolved from a single drive into a RAID with multiple disks, the interface remained simple, but the RAID itself did not. The result is a system full of misinformation: the file system no longer has an accurate model of disk behavior, and the now-complex storage system does not have a good understanding of what to expect from the file system.

EXRAID and I-LFS bridge this information gap by design: the presence of multiple regions is exposed directly to the file system, enabling new functionality. In this paper, we have explored the implementation of on-line expansion, dynamic parallelism, flexible redundancy, and lazy mirroring in I-LFS. All were implemented in a relatively straight-forward manner within the file system, increasing system manageability, performance, and functionality, while maintaining a reasonable level of overall system complexity. Some of these aspects of I-LFS would be difficult if not impossible to build in the traditional storage protocol stack, highlighting the importance of implementing functionality in the correct layer of the system.

Though we have chosen a single point in the design space of storage protocol stacks, other arrangements are possible and perhaps even preferable; we hope that they will be explored. Whatever the conclusion of research on the division of labor between file and storage systems, we believe that the proper division should be arrived upon via design, implementation, and thorough experimentation, not via historical artifact.

## 10 Acknowledgements

We would like to thank our shepherd, Elizabeth Shriver, as well as John Bent, Nathan Burnett, Brian Forney, Florentina Popovici, Muthian Sivathanu, and the anonymous reviewers for their excellent feedback.

This work is sponsored by NSF CCR-0092840, CCR-0098274, NGS-0103670, CCR-0133456, ITR-0086044, and the Wisconsin Alumni Research Foundation. Timothy E. Denehy is sponsored by an NDSEG Fellowship from the Department of Defense.

## References

- [1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, and R. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 109–26, Copper Mountain Resort, CO, December 1995.
- [2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.
- [3] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1997 ACM SIGMOD Conference on the Management of Data (SIGMOD '97)*, pages 243–254, Tucson, AZ, May 1997.
- [4] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *The 1999 Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, Atlanta, GA, May 1999.
- [5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 33–40, Schloss Elmau, Germany, May 2001.
- [6] T. Bray. The Bonnie File System Benchmark. <http://www.textuality.com/bonnie/>.
- [7] A. Brown and D. A. Patterson. Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 263–276, San Diego, CA, June 2000.
- [8] D. Comer. *Internetworking with TCP/IP Vol. 1: Principles, Protocols and Architecture*. Prentice Hall, London, 2 edition, 1991.
- [9] T. H. Cormen and D. Kotz. Integrating Theory And Practice In Parallel File Systems. In *Proceedings of the 1993 DAGS/PC Symposium (The Dartmouth Institute for Advanced Graduate Studies)*, pages 64–74, Hanover, NH, June 1993.
- [10] T. Cortes and J. Labarta. Extending Heterogeneity to RAID level 5. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.
- [11] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 15–28, Asheville, NC, December 1993.
- [12] D. R. Engler and M. F. Kaashoek. Exterminate All Operating System Abstractions. In *The Fifth Workshop on Hot Topics in Operating Systems (HotOS V)*, Orcas Island, WA, May 1995.
- [13] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 237–252, San Francisco, CA, January 1992.
- [14] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File Server Scaling with Network-Attached Secure Disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 272–284, Seattle, WA, June 1997.
- [15] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 29–43, Asheville, NC, December 1993.

- [16] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference*, Berkeley, CA, January 1994.
- [17] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFS: A High Performance Portable Parallel File System. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, Spain, July 1995.
- [18] V. Jacobson. How to Kill the Internet. <ftp://ftp.ee.lbl.gov/talks/vj-webflame.ps.Z>, 1995.
- [19] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Summer. One-level Storage System. *IRE Transactions on Electronic Computers*, EC-11:223–235, April 1962.
- [20] B. W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 33–48, Bretton Woods, NH, December 1983. ACM.
- [21] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 84–92, Cambridge, MA, October 1996.
- [22] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 238–251, Saint-Malo, France, October 1997.
- [23] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.
- [24] N. Nieuwejaar and D. Kotz. The Galley Parallel File System. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.
- [25] W. Norcutt. The IOzone Filesystem Benchmark. <http://www.iozone.org/>.
- [26] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, IL, June 1988.
- [27] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Comm. Assoc. Comp. Mach.*, 17(7):365–375, July 1974.
- [28] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, San Diego, CA, June 2000.
- [29] D. Roselli, J. N. Matthews, and T. E. Anderson. File System Fingerprinting. Works-In-Progress at the Third Symposium on Operating Systems Design and Implementation (OSDI '99), February 1999.
- [30] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
- [31] D. S. H. Rosenthal. Evolving the Vnode Interface. In *Proceedings of the 1990 USENIX Summer Technical Conference*, pages 107–118, Anaheim, CA, 1990.
- [32] J. R. Santos and R. Muntz. Performance Analysis of the RIO Multimedia Storage System with Heterogeneous Disk Configurations. In *ACM Multimedia '98*, December 1998.
- [33] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding When To Forget In The Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 110–123, Kiawah Island Resort, SC, December 1999.
- [34] S. Savage and J. Wilkes. AFRAID — A Frequently Redundant Array of Independent Disks. In *Proceedings of the 1996 USENIX Technical Conference*, pages 27–39, San Diego, CA, January 1996.
- [35] J. Schindler and G. R. Ganger. Automated Disk Drive Characterization. Technical Report CMU-CS-99-176, Carnegie Mellon, 1999.
- [36] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, January 2002.
- [37] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the 1993 USENIX Winter Technical Conference*, pages 307–326, San Diego, CA, January 1993.
- [38] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File System Logging versus Clustering: A Performance Comparison. In *Proceedings of the 1995 USENIX Annual Technical Conference*, pages 249–264, New Orleans, LA, January 1995.
- [39] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 71–84, San Diego, CA, June 2000.
- [40] D. Stodolsky, M. Holland, W. V. Courtright II, and G. A. Gibson. Parity-logging disk arrays. *ACM Transactions on Computer Systems*, 12(3):206–235, August 1994.
- [41] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, January 1996.
- [42] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.
- [43] D. Teigland. The Pool Driver: A Volume Driver for SANs. Master's thesis, University of Minnesota, December 1999.
- [44] D. Teigland and H. Mauelshagen. Volume Managers in Linux. In *FREENIX Track of the USENIX 2001 Annual Technical Conference*, Boston, MA, June 2001.
- [45] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237, Saint-Malo, France, October 1997.
- [46] T. Ts'o. <http://e2fsprogs.sourceforge.net/ext2.html>, June 2001.
- [47] R. van Renesse. Masking the Overhead of Protocol Layering. In *Proceedings of the ACM SIGCOMM '96 Conference*, pages 96–104, Palo Alto, CA, 1996.
- [48] Veritas. <http://www.veritas.com>, June 2001.
- [49] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, LA, February 1999.
- [50] J. Wilkes. DataMesh Research Project, Phase I. In *Proceedings of the USENIX File Systems Workshop*, pages 63–69, May 1992.
- [51] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.
- [52] R. Zimmermann and S. Ghandeharizadeh. HERA: Heterogeneous Extension of RAID. Technical Report USC-CS-TR98-685, University of Southern California, 1998.

# Maximizing Throughput in Replicated Disk Striping of Variable Bit-Rate Streams

Stergios V. Anastasiadis\*  
Department of  
Computer Science  
Duke University  
stergios@cs.duke.edu

Kenneth C. Sevcik  
Department of  
Computer Science  
University of Toronto  
kcs@cs.toronto.edu

Michael Stumm  
Dept of Electrical and  
Computer Engineering  
University of Toronto  
stumm@eecg.toronto.edu

## Abstract

*In a system offering on-demand real-time streaming of media files, data striping across an array of disks can improve load balancing, allowing higher disk utilization and increased system throughput. However, it can also cause complete service disruption in the case of a disk failure. Reliability can be improved by adding data redundancy and reserving extra disk bandwidth during normal operation. In this paper, we are interested in providing fault-tolerance for media servers that support variable bit-rate encoding formats. Higher compression efficiency with respect to constant bit-rate encoding can significantly reduce per-user resource requirements, at the cost of increased resource management complexity. For the first time, the interaction between storage system fault-tolerance and variable bit-rate streaming with deterministic QoS guarantees is investigated. We implement into a prototype server and experimentally evaluate, using detailed simulated disk models, alternative data replication techniques and disk bandwidth reservation schemes. We show that with the minimum reservation scheme introduced here, single disk failures can be tolerated at a cost of less than 20% reduced throughput during normal operation, even for a disk array of moderate size. We also examine the benefit from load balancing techniques proposed for traditional storage systems and find only limited improvement in the measured throughput.*

## 1 Introduction

Striping media files across multiple disks has the advantage of keeping the disks implicitly load-balanced.

\*Part of this research has been done, while the co-author was PhD student at the University of Toronto. Financial support from the Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged.

With several concurrent sessions of playback asynchronously started, different parts of each file are accessed from different disks. As a result it is no longer necessary to replicate media files according to their popularity, which leads to lower resource requirements and system administration cost. The disadvantage of disk striping is decreased system reliability because media files are left partially inaccessible when one or more disks fail. This causes service disruption to all the users served by the disk array at the moment of the failure. In contrast, when an entire file is stored on a single disk, only those users accessing files on the failed disk are negatively affected.

Previous work has addressed the general problem of disk array reliability by using data redundancy techniques to allow recovery of inaccessible data [12]. Some of these techniques have been successfully extended for handling the case of striped media files as well [6, 25]. However, all the known analytical and experimental work on this subject is either limited to streams of constant bit rates (CBR), or assumes stochastic admission control [27, 29].

Variable bit-rate (VBR) encoding of video can considerably reduce the size of the generated media files when compared to constant bit-rate encoding of equivalent perceptual quality [18, 22]. In addition, knowledge about the resource requirements of stored streams during transmission can be leveraged for better predicting access delays, and offering deterministic QoS guarantees [11]. Although striping of VBR streams has been previously studied, it remains unclear whether increased reliability can be provided with deterministic QoS guarantees in cost-effective ways.

Variability in the resource requirements over time makes efficient disk space allocation combined with access delay predictability a challenging task. Data striping across multiple disks with sufficient redun-



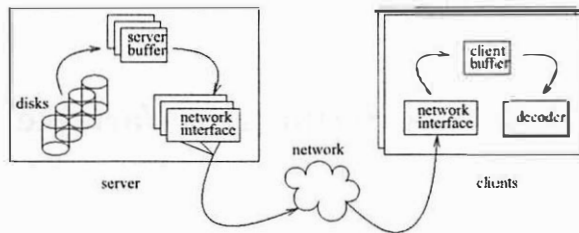


Figure 1: Compressed video streams are stored across multiple disks of the media server. Multiple clients (or proxies) can connect and start playback sessions via separate network links.

dancy to tolerate failures further aggravates the problem due to the need for keeping balanced the storage space and bandwidth requirements across the disk array, under both normal-operation and failed-disk conditions. In the present paper, we describe a number of data redundancy and bandwidth reservation schemes that can tolerate single disk failures without service interruption. We experimentally evaluate the cost of increased reliability in terms of reduced throughput during the normal operation of the system using a video server prototype implementation and MPEG-2 streams. We also investigate the achieved disk bandwidth utilization during normal and failed-disk operation. Additionally, we examine the extra benefit from retrieving data replicas stored on the least loaded disks, and from fragmenting data replicas across multiple disks.

The rest of this paper is organized as follows. In Section 2, we describe basic assumptions and architectural decisions of our system. In Sections 3, 4 and 5, we introduce alternative policies for replicating stream data and reserving disk bandwidth for improved reliability. In Section 6, we briefly present our prototype implementation and the experimentation environment that we use. In Section 7, we compare the performance of different replication techniques under alternative bandwidth reservation schemes and load balancing enhancements. In Section 8, we discuss possible improvements and extensions, and in Section 9 we summarize our conclusions.

## 2 System Architecture

In the present section, we describe the system architecture along with important resource management and reservation techniques that we use [2].

### 2.1 Overview

The operation of our media server is typical in current system designs. Client devices submit playback requests concurrently to the server. The system is assumed to operate according to the server-push model. When a playback session starts, the server periodically sends data to the client until either the end of the stream is reached, or the client explicitly requests suspension of the playback. Data transfers occur in rounds of fixed duration  $T_{round}$ . In each round, an appropriate amount of data is retrieved from the disks into a set of server buffers reserved for each active client. Concurrently, data are sent from the server buffers to the client through the network interfaces (Figure 1). The amount of stream data periodically sent to the client is determined by the decoding frame rate of the stream, the buffering constraints of the receiver, and the resource management policy of the network. As a minimum requirement, the client should receive in each round the amount of data that will be needed by the decoder during the next round.

The streams are compressed according to any encoding scheme that supports constant quality quantization parameters and variable bit rates. Playback requests arriving from the clients are initially directed to an admission control module, where it is determined whether enough resources exist to activate the requested playback session either immediately or within a limited number of rounds. A schedule database maintains for each stream information on how much data needs to be accessed from each disk in any given round, the amount of server buffer space required, and how much data needs to be transferred to the client. This scheduling information is generated when the media stream is first stored, and is used for both admission control and transfer of data during playback.

### 2.2 Stride-Based Disk Space Allocation

In our experiments, we use a method called *stride-based allocation* for allocating disk space [3]. In stride-based allocation, disk space is allocated in large, fixed-sized chunks called *strides*. The strides are chosen larger than the maximum stream request size per disk during a round. This size is known a priori, since stored streams are accessed sequentially according to a predefined (generally variable) rate. A stride may contain data of more than one round. When a stream is retrieved, only the requested amount of data is fetched to memory during a round, and not the entire stride.

Stride-based allocation eliminates external fragmentation due to the fixed-size strides. Internal fragmentation remains negligible because of the large size of the streams relative to strides. Another advantage of stride-based allocation is that it sets an upper-bound on the estimated disk access overhead during retrieval. Since the size of a stream request never exceeds the stride size during a round, at most two partial stride accesses will be required to serve the request of a round on each disk.

### 2.3 Reservation of Server Resources

A mathematical abstraction of the resource requirements is necessary for scheduling streams. We consider a system with  $D$  functionally equivalent disks. Data of each stream are stored as sequences of strides on each disk. Each stride comprises an integer number of consecutive logical blocks with fixed size  $B_l$ . The logical block size is a multiple of the physical sector size  $B_p$  of the disk. Both the disk transfer requests and the memory buffer reservations are specified in multiples of the block size  $B_l$ . The *Disk Striping Sequence*  $S_d$  of length  $L_d$  determines the amount of data  $S_d(i, k)$ ,  $0 \leq i \leq L_d - 1$ , that are retrieved from disk  $k$ ,  $0 \leq k \leq D - 1$ , in round  $i$ .

We assume that each disk has edge to edge seek time  $T_{fullSeek}$ , single-track seek time  $T_{trackSeek}$ , average rotational latency  $T_{avgRot}$ , and minimum internal transfer rate  $R_{disk}$ . The stride-based disk space allocation policy enforces an upper bound of at most two disk arm movements per disk for each client per round. The total seek distance can also be limited using a Circular SCAN disk scheduling policy. Let  $M_i$  be the number of active streams during round  $i$  of the system operation, and  $l_j$  the round of system operation that the playback of stream  $j$ ,  $1 \leq j \leq M_i$ , started. Then, the total access time on disk  $k$  in round  $i$  of the system operation can be approximated by the following expression:

$$T_{disk}(i, k) = 2T_{fullSeek} + 2M_i \cdot (T_{trackSeek} + T_{avgRot}) + \sum_{j=1}^{M_i} S_d^j(i - l_j, k) / R_{disk}$$

where  $S_d^j$  is the disk striping sequence of client  $j$ . The parameter  $T_{fullSeek}$  is counted twice due to the disk arm movement from the C-SCAN policy, while the factor two in the second term is due to the stride-based allocation scheme we use. The first term should be accounted for only once in the time reservation of each disk, but each client  $j$  incurs an

extra access time of

$$T_{disk}^j(i, k) = 2 \cdot (T_{trackSeek} + T_{avgRot}) + S_d^j(i - l_j, k) / R_{disk}$$

on disk  $k$  during round  $i$ , when  $S_d^j(i - l_j, k) > 0$ , and zero otherwise. Reservations of network bandwidth and buffer space are more straightforward, and based on the network and buffer sequence of each accepted playback request, respectively.

### 2.4 Variable-Grain Striping

For striping stream data across multiple disks, we use the *Variable-Grain Striping* policy. Data consumed by a client during a playback round is stored on (and accessed from) a single disk, while different disks are visited in round-robin fashion during successive rounds of a stream playback. When compared against alternative striping techniques, variable-grain striping demonstrates significant performance advantage due to i) reduced disk access overhead from accessing at most one disk per stream in a round, and ii) improved disk bandwidth utilization by statistically multiplexing I/O requests of different sizes from concurrently served streams [2].

## 3 Data Redundancy Policies

Due to the large number of components involved, it is necessary to assume device failures during the lifetime of a typical commercial server installation. With the estimated Mean Time To Failure of a modern disk at about  $MTTF_{disk} = 1,200,000$  hours, combining  $D = 1024$  disks results in  $MTTF_{array} = \frac{MTTF_{disk}}{D} = 49$  days, assuming failure independence among different devices [26].<sup>1</sup> A typical system with 1024 drives could support about 51,000 concurrent playbacks of 5 Mbit/s average bit rate each.<sup>2</sup> Although the disks are likely to be distributed across multiple independent servers, building a single large disk array from distributed components has also been demonstrated in the past for media streaming services [9, 19].

In order to provide higher system reliability, data redundancy techniques can be used. However, a practical solution should minimize the extra computa-

<sup>1</sup>The calculation assumes Seagate Cheetah 18GB Ultra160 SCSI disks with 31 MB/s formatted minimum internal transfer rate [28].

<sup>2</sup>These numbers are realistic in light of the popularity of similar services. For example, the number of cable television subscribers in 1998 was estimated to exceed 65 million in the US alone [13].

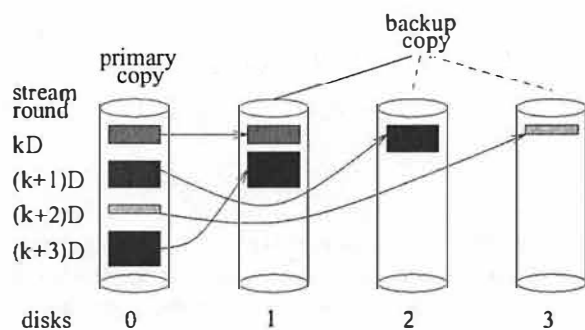


Figure 2: **Deterministic Replica Placement.** Data of a media file stored consecutively on disk 0 and retrieved during different playback rounds are replicated round-robin across the other disks. The primary data of the other disks are replicated in a similar way.

tion, storage and bandwidth requirements with respect to the non-redundant case. The present study focuses on single disk failures which are the most common. Multiple disk failures are less likely to occur simultaneously [12], and possible ways for handling them are described briefly later.

In the past, several parity-based techniques have been proposed that store error-correcting code for the data blocks of different disks [12]. When a disk fails, redundant information available on the surviving disks is used to recover the missing data blocks. Parity-based techniques trade extra disk bandwidth or memory buffer requirements for reduced storage space. Since disk storage space currently has the lowest cost of the three resources, it has been suggested that replication rather than parity is the preferred technique for tolerating disk failures [10, 17]. Furthermore, implementation of parity-based data recovery in a distributed architecture requires additional data traffic among different nodes [9]. This can introduce significant extra complexity and resource requirements in terms of network bandwidth and buffer space. For the above reasons, we do not consider parity-based techniques any further here.

With mirroring techniques, the data of each disk are replicated on one or more different disks. We refer to the original copy of the data as *primary* and the additional copy as *backup*. Although the two copies can be used symmetrically, distinct placement policies can be applied to each of them as we describe shortly. When one disk fails, its data remain available by retrieving their backup replicas from the rest of the disks. The required storage space is roughly doubled and the needed bandwidth from each disk is at most twice that of the non-redundant case. The

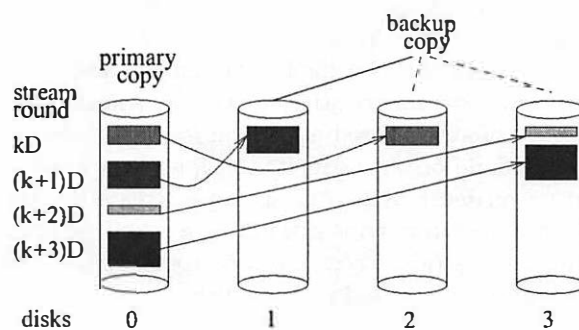


Figure 3: **Random Replica Placement.** Data of a media file stored consecutively on disk 0 and retrieved during different playback rounds are replicated on randomly chosen disks 1 to 3. The primary data copies of disks 1 to 3 are replicated in a similar way.

backup replica of each data block can be stored in its entirety on a different disk, which requires only one access in the case of failure and minimizes the access overhead. The alternative of declustering a backup replica across multiple disks can potentially better balance the extra access load, but incurs the additional overhead of multiple accesses in the case of a disk failure.

### 3.1 Deterministic Replica Placement

In previous work, we have demonstrated that variable-grain striping of media files leads to equally utilized disks under sequential playback workloads [2]. Although mirroring has previously been only used with data striped using fixed-size blocks, in principle it could be applied to variable-grain striping as well. During sequential playback of a media file with no failed disks, each disk is accessed every  $D$  rounds, where  $D$  is the total number of disks. In order to preserve the load-balancing property when a disk fails, data of a media file stored consecutively on each disk could be replicated round-robin across the remaining disks (or a subset of them). The unit of replication corresponds to data retrieved by a client during one round of playback. We call this mirroring approach *Deterministic Replica Placement*. In Figure 2, for example, disk 0 is shown to store stream data requested during rounds  $k \cdot D$ ,  $(k+1) \cdot D$ ,  $(k+2) \cdot D$  and  $(k+3) \cdot D$ . The respective replicas are distributed round-robin among disks 1, 2 and 3.

### 3.2 Random Replica Placement

Intuitively, having replicas of one disk's primary data distributed round-robin across the rest of the disks can keep the surviving disks equally utilized when one disk fails. An alternative replication approach

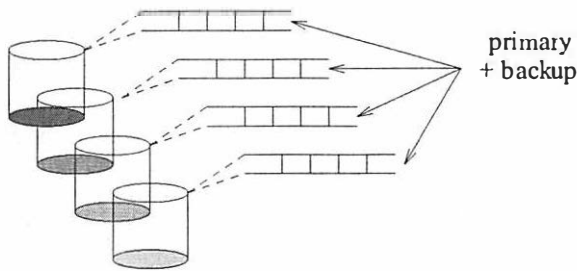


Figure 4: **Mirroring Reservation.** For each disk, there is a separate vector indexed by round number that accumulates the total estimated access time for retrieving primary and backup data in each round.

would use some pseudo-random sequence for specifying the disks that store the backup copies of one disk's primary data. An obvious constraint is that primary and backup copies are stored on different devices. The unit of replication corresponds to the data of a media file requested by a client in one round of playback. We call this mirroring technique *Random Replica Placement*.

An example is shown in Figure 3, where backup copies of data requested in rounds  $k \cdot D, \dots, (k + 3) \cdot D$  are randomly placed on disks 1 to 3. It has been previously suggested that random placement of primary and backup replicas across different disks is applicable to a wider range of workload types and can outperform striping policies with round-robin placement [27]. In a later section, we examine this argument in the particular case of variable bit-rate streams by comparing it against deterministic replica placement.

## 4 Disk Bandwidth Reservation

Our goal in this section is to allocate resources in such a manner that service to accepted requests will not be interrupted during (single) disk failures. Retrieving backup replicas of data stored on a failed disk requires extra bandwidth to be reserved in advance across the surviving disks. This implies that the system will normally have to operate below full capacity. Alternatively, when a disk fails and no extra bandwidth has been reserved, service will become unavailable for a number of active users with aggregate bandwidth requirements no less than the transfer capacity of one disk, assuming that data have been replicated as described previously.

The net benefit from uninterrupted service during disk failures is equal to the difference between two measures. One is the cost of having users frustrated

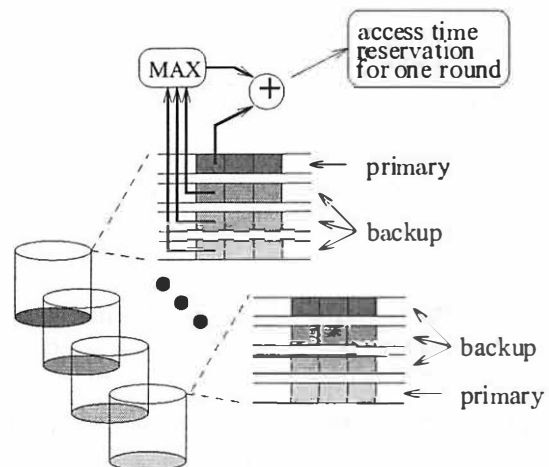


Figure 5: **Minimum Reservation.** For each disk, we maintain  $D$  separate vectors indexed by round number. One of them accumulates access delays for retrieving primary data. The remaining  $D - 1$  vectors accumulate access delays for retrieving backup replicas that correspond to primary data stored on each of the other  $D - 1$  disks. In each round, the sum of the primary data access time and the maximum of the backup data access times is reserved on each disk.

due to interrupted service from a failed disk. Its quantification would require determining the minimum number of users negatively affected when a disk fails. Detailed study of this issue is left for future work. The other measure is the cost of rejecting user requests due to additionally reserved disk bandwidth during normal operation. In the rest of this paper, we describe alternative approaches for reserving disk bandwidth (or equivalently access time) and improving reliability in media servers that support variable bit-rate streams. Subsequently, we experimentally evaluate the actual cost of these approaches in terms of reduced system throughput during normal system operation.

In what we call *Mirroring Reservation*, disk bandwidth is reserved for both the primary and backup replicas of a media file during its playback (Figure 4). At first glance, this seems to be a reasonable approach for guaranteeing timely access to backup replicas during a single disk failure. However, when compared to the non-redundant case, it doubles the bandwidth requirements of each stream and halves the maximum system throughput, assuming disk bandwidth is the bottleneck resource in the system. Indeed, we would prefer that the load normally handled by a failed disk is equally divided among the  $D - 1$  surviving disks. Thus, tolerating one disk failure should require that the extra

Replica Placement	Disk Bandwidth Reservation	
	Mirroring	Minimum
Deterministic	✓	✓
Random	✓	✓

Table 1: The replica placement policies can be orthogonally combined with the disk bandwidth reservation schemes.

bandwidth reserved on each disk be equal to  $\frac{1}{D-1}$  its bandwidth capacity. Instead, mirroring reservation reserves extra bandwidth on each disk equal to half its bandwidth capacity.

Essentially, it is wasteful to reserve on a disk extra bandwidth for accessing backup replicas of primary data stored on more than one other disk. When a disk fails, we only need an estimate of the additional access load incurred on every surviving disk. In order to know that, the access time of the backup replicas stored on one disk can be accumulated separately for every disk that stores the corresponding primary data. Then, the additional access time that has to be reserved on a disk in each round is equal to the maximum time required for retrieving backup replicas for another disk that has failed. The maximum across every other disk is reserved, since we don't know in advance which other disk is going to fail.

For each disk, our implementation maintains  $D$  vectors, indexed by round number of system operation. One of the vectors keeps track of the total access time required for retrieving primary data. The remaining  $D-1$  vectors keep track of access delays due to backup data corresponding to primary data of the remaining disks. For every disk, we reserve the sum of the primary data access time and the maximum of the backup data access times required in each round. We refer to this more efficient scheme as *Minimum Reservation*. An example with four disks is illustrated in Figure 5. In a later section, we discuss ways for limiting the additional computational and memory requirements of this approach.

The two disk bandwidth reservation schemes that we just described can be orthogonally combined with the two replica placement policies that we introduced previously, as it is shown in Table 1.

## 5 Load Balancing Enhancements

The load of a failed disk could possibly be shared more fairly among the surviving disks if each backup replica was declustered across multiple devices. Therefore, we break each backup replica into blocks of

fixed size  $B_d$ , and we call this load balancing technique *Backup Replica Declustering*. We choose  $B_d$  to be an integer multiple of the logical block size  $B_l$ , introduced previously. We allow the last fragment of the replica to have a size that is smaller than  $B_d$  but integer multiple of  $B_l$ .

The backup replica blocks corresponding to the primary data of each disk are distributed either round-robin or pseudo-randomly across the rest of the disks, depending on whether deterministic or random replica placement is used. In the case of random replica placement, we improve block distribution by avoiding reusing the same disk for storing multiple replica blocks of the same file in one round unless we are running out of disks. When multiple blocks of size  $B_d$  are retrieved from a disk during one round of a file playback, the minimum required number of read requests is submitted to the disk, instead of one per block.

Alternatively, during normal operation we could take advantage of multiple available data replicas by dynamically deciding to retrieve the replica stored on the disk expected to be the least loaded. The disk choice could be based on access time estimations available through resource reservations that are made during admission control. We use the term *Dynamic Balancing* for this technique. It can be fully applied when all the disks are functional and is expected to reduce the load of the most heavily utilized disks in each round.

Both these two techniques have previously been found to improve performance when applied to traditional transaction processing workloads [23]. Replica declustering has also been tried with constant bit-rate stream playback [9, 15]. Due to the potential for load imbalance and reduced device utilization introduced by variable bit-rate streams, we investigate the benefit from load-balancing techniques in that context.

## 6 Experimentation Environment

In order to keep our presentation complete, we briefly describe here important aspects of our prototype implementation, the characteristics of our benchmarks, and the performance evaluation method that we use for our experiments [2].

### 6.1 Prototype Overview

We have designed and built a media server experimentation platform, in order to evaluate the resource requirements of alternative disk replication policies [2]. The different modules are implemented in about 17,000 lines of C++/Pthreads code on AIX4.1. The



Seagate Cheetah ST-34501N	
Data Bytes per Drive	4.55 GB
Average Sectors per Track	170
Data Cylinders	6,526
Data Surfaces	8
Zones	7
Buffer Size	512KB
Track to Track Seek(read/write)	0.98/1.24 msec
Maximum Seek(read/write)	18.2/19.2 msec
Average Rotational Latency	2.99 msec
Internal Transfer Rate	
Inner Zone to Outer Zone Burst	122 to 177 Mbit/s
Inner Zone to Outer Zone Sustained	11.3 to 16.8 MB/s

Table 2: Features of the Seagate SCSI disk assumed in our experiments.

code is linked either to the University of Michigan DiskSim disk simulation package [16], which incorporates advanced features of modern disks such as on-disk cache and zones for simulated disk access time measurements, or to hardware disks through their raw device interfaces. The indexing metadata are stored as regular Unix files, and during operation are kept in main memory.

The basic responsibilities of the media server include file naming, resource reservation, admission control, logical to physical metadata mapping, buffer management, and disk and network transfer scheduling.

With appropriate configuration parameters, the system can operate at different levels of detail. In *Admission Control* mode, the system receives playback requests, does admission control and resource reservation, but no actual data transfers take place. In *Simulated Disk* mode, most modules become functional and disk request processing is simulated using the specified DiskSim disk array. Techniques for file system simulation similar to those previously proposed are used for integrating the simulated disks with our media server prototype [31]. There is also the *Full Operation* mode, where the system accesses hardware disks and transfers data to fixed client network addresses. For the experiments in the current study, we used both the Admission Control and the Simulated Disk Mode.

## 6.2 Performance Evaluation Method

We assume that playback initiation requests arrive independently of one another, according to a Poisson process. The system load can be controlled through the arrival rate  $\lambda$  of playback initiation requests. Assuming that the disk transfers are the bottleneck, we consider a “perfectly efficient system” that incurs no disk overhead when accessing data. Then, we choose the maximum arrival rate  $\lambda = \lambda_{max}$  of playback re-

Content Type	Avg Bytes per Round	Max Bytes per Round	CoV per Round
Science Fiction	624,935	1,201,221	0.383
Music Clip	624,728	1,201,221	0.366
Action	624,194	1,201,221	0.245
Talk Show	624,729	1,201,221	0.234
Adventure	624,658	1,201,221	0.201
Documentary	625,062	625,786	0.028

Table 3: We used six MPEG-2 video streams of 30 minutes duration each. The coefficient of variation shown in the last column changes according to the content type.

quests equal to the mean stream completion rate in that perfectly efficient system. This creates enough system load to show the performance benefit of arbitrarily efficient data striping policies. The mean stream completion rate  $\mu$ , expressed in streams per round, for streams of average data size  $S_{tot}$  bytes becomes:

$$\mu = \frac{D \cdot R_{disk} \cdot T_{round}}{S_{tot}} \frac{streams}{round}. \quad (1)$$

The corresponding system load becomes:  $\rho = \frac{\lambda}{\mu} \leq 1$ , where  $\lambda \leq \lambda_{max} = \mu$ .

For each playback request that arrives, the admission control module checks whether available resources exist for every round during playback. The test considers the data transfer requirements of the requested playback for every round and also the corresponding available disk transfer time, network transfer time and buffer space in the system. If the request cannot be initiated in the next round, the test is repeated for each round up to  $\lceil \frac{1}{\lambda} \rceil$  rounds into the future, until the first round is found where the requested playback can be started with guaranteed sufficiency of resources. Checking  $\lceil \frac{1}{\lambda} \rceil$  rounds into the future achieves most of the potential system capacity as was shown previously [2]. If not accepted, the request is rejected rather than being kept in a queue.

## 6.3 Experimentation Setup

We used six different VBR MPEG-2 streams of 30 minutes duration each. Every stream has 54,000 frames with a resolution of 720x480 and 24 bit color depth, 30 frames per second frequency, and a  $1B^2PB^2PB^2PB^2PB^2$  15 frame Group of Pictures structure. The encoding hardware that we use generates bit rates between 1 Mbit/s and 9.6 Mbit/s. The statistical characteristics of the clips are given in Table 3. The coefficients of variation of bytes per round lie between 0.028 and 0.383, depending on the content type. In the *mixed* benchmark, the six different

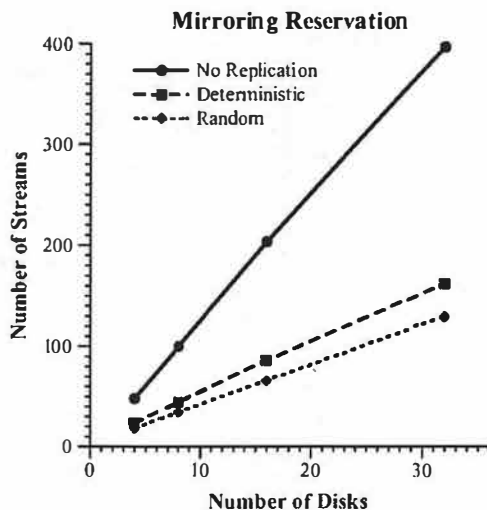


Figure 6: The mirroring reservation scheme reduces the number of streams supported by a factor of two compared to the no-replication case. Deterministic replica placement sustains an advantage of 25% or more relative to random replica placement under the mixed stream workload.

streams are submitted round-robin. Where appropriate, experimental results from individual stream types are also shown.

The disks assumed in our experiments are Seagate Cheetah with ultra-wide SCSI interface and the features shown in Table 2. Such disks were state of the art about three years ago, and have all the basic architectural characteristics of today's high-end drives. The logical block size  $B_l$  was set to  $16KB$  bytes, while the physical sector size  $B_p$  was equal to 512 bytes. The stride size  $B_s$  in the disk space allocation was set to 2 MB. The server memory is organized in buffers of fixed size  $B_l = 16KB$  bytes each, with space of 64 MB for every extra disk. The available network bandwidth was assumed to be infinite, leaving contention for the network outside the scope of the current work.

In our experiments, the round time was set equal to one second. We found this round length to achieve most of the system capacity with tolerable initiation latency. This choice also facilitates comparison with previous work in which one second rounds were used. We used a warmup period of 3,000 rounds and calculated the average number of active streams from round 3,000 to round 9,000. The measurements were repeated until the half-length of the 95% confidence interval was within 5% of the estimated mean value of the number of active streams. The system load was fixed at  $\rho = 80\%$ , which allows the system to reach its capacity while keeping the playback startup

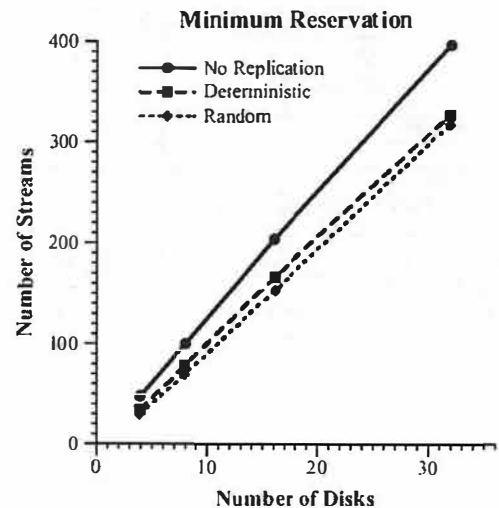


Figure 7: With minimum reservation and number of disks increasing from 4 to 32, the throughput advantage of deterministic over random replica placement drops from 15% to 3%. The corresponding throughput disadvantage of deterministic placement with respect to no replication drops from 28% to 17%.

latency limited [2].

## 7 Experimental Evaluation

We compare the data replication and bandwidth reservation techniques that we introduced with respect to the average number of active playback sessions that can be supported by the server. The objective is to make this number as high as possible. We provide supplementary performance intuition with statistics on reserved and utilized disk access time across different stream types and numbers of disks.

We start with a performance comparison between the deterministic and random replica placement policies under the mirroring reservation scheme. Subsequently, we examine the improvement to the two placement policies when minimum reservation is applied. We also investigate the benefit of dynamic balancing assuming that disk bandwidth in each round is reserved for only one data replica out of the two available. Finally, we consider declustering the backup replica of each stream across multiple disks and allocating bandwidth according to the minimum reservation scheme.

### 7.1 Replica Placement Comparison

We use the mixed stream workload to compare the performance of alternative replica placement policies under the mirroring reservation scheme (Figure 6). With the number of disks varying between 4 and 32, the measured throughput of replicated disk striping

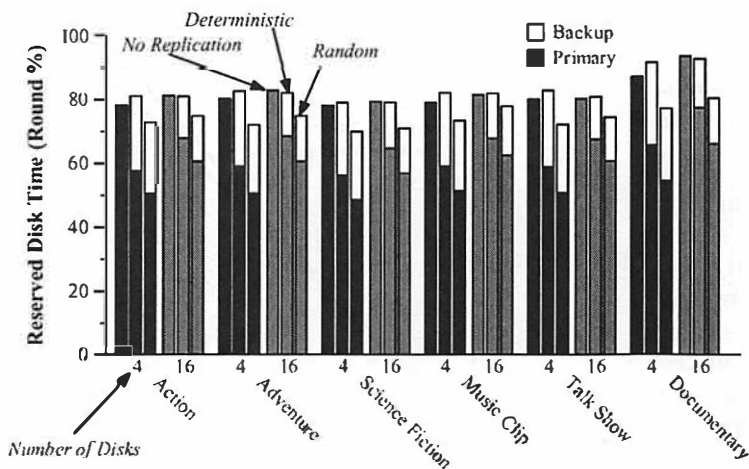


Figure 8: The disk time reserved in each round when individual stream types are used with minimum reservation. With deterministic replica placement, the disk time reserved for backup accesses drops from about 24% to 14% of the round length as the number of disks increases from 4 to 16. With random replica placement, the respective percentage drops from about 22% to 14%.

is less than half of what is achieved with no replication. In addition, deterministic replica placement achieves a throughput advantage of 25% or more relative to random replica placement.

From measurements that we did (not shown here), we found that about half of the average disk time reserved in the replicated case is wasted for the possibility that the backup data will be retrieved. Furthermore, the access time reserved on each disk by random replica placement is about 15%-25% less than that of deterministic placement. Pseudo random choice of the disk that stores a backup replica does not completely eliminate the possibility of one disk storing more replicas than another, especially with small disk arrays. The probability of that occurring drops as the size of the disk array increases, though. However, deterministic placement is more consistent in fairly distributing the access load across the disk array devices.

When a disk fails, about 25-30% of the reserved disk bandwidth remains unused under both placement policies. This is not surprising, since the mirroring reservation scheme allocates disk bandwidth for both the primary and backup replicas of each accepted stream. We see how this inefficiency is alleviated by the minimum reservation scheme in the following subsection.

## 7.2 Minimizing Reserved Bandwidth

The minimum reservation scheme improves disk utilization by allocating on each disk the extra time re-

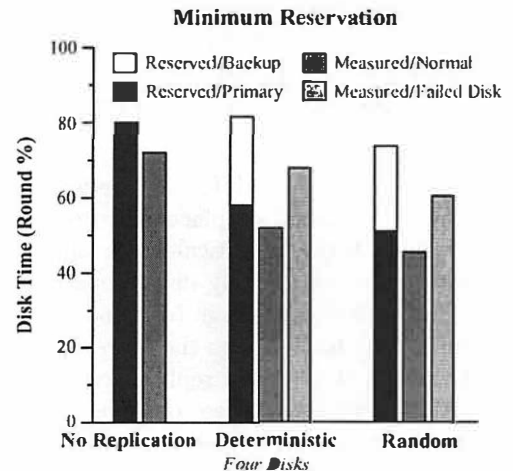


Figure 9: During normal operation, the average disk access time that is measured in each round remains within 6-8% below the time reserved for primary data accesses. When a disk fails, the measured access time is 13-14% lower than the total reserved.

quired for accessing backup replicas of only one other disk. In order to ensure that any single disk failure can be handled properly, each disk keeps track and reserves the maximum additional time required for handling potential failure of any other disk. This maximum requirement is calculated separately and is generally expected to be different for each disk in each round.

Figure 7 compares the throughput of the different replica placement policies under minimum reservation. At eight disks, the number of streams supported by deterministic placement is only 21% lower than that with no replication. This difference becomes 18% and 17%, respectively, with sixteen and thirty two disks. From the way that the disk bandwidth is allocated in the minimum reservation scheme, we would expect the total bandwidth that remains unutilized during normal operation to be equal to the bandwidth capacity of one disk. Therefore, the percentage of unused throughput with respect to the non-replicated case should be decreasing proportionally with the number of disks in the system. For example, ideally with 16 disks only the  $\frac{1}{16} = 6.25\%$  of the total disk bandwidth should remain unused during normal operation.

However, in practice, the percentage of the total unused bandwidth of each disk does not change proportionally with the number of disks (Figures 8). This effect can be explained by the *MAX()* operator that is applied over the estimated time for accessing the backup replicas of different disks, in combina-

tion with the relatively large size (more than half megabyte on average) of the data retrieved for a stream in each round. We explore later the potential improvement from declustering backup replicas across multiple disks.

Additionally, the difference between deterministic and random replica placement becomes less significant than the statistical uncertainty at thirty two disks. Not surprisingly, deterministic placement maintains a clear advantage (of about 15%) for smaller disk array sizes due to the more regular way of distributing the backup replica access load across the different devices. These observations are consistent with the average access time reserved on each disk across different stream types and disk array sizes shown in Figure 8.

In Figure 9, we show the measured disk busy time. Under normal disk operation, we observe that deterministic placement keeps the disks busy an amount of time that is 6% lower than what is reserved for primary data.<sup>3</sup> When one disk fails, the remaining disks are busy 14% time less than the total reserved. With random replica placement, the corresponding difference becomes 13% of the round length. This is a significant improvement in comparison to the 25-30% difference between reserved and measured time that we reported for mirroring reservation. We should keep in mind that, with disk array size equal to four, about one third of each disk's bandwidth has to be reserved for the case that one disk fails, and this fraction drops as the disk array size increases to sixteen (Figure 8).

It is interesting that, when the reserved backup access time is put into use due to a disk failure, the difference between reserved and utilized access time increases from 6-8% to 13-14%. At first glance this discrepancy appears as reduced accuracy in access time estimation. In fact it is due to the MAX() operator that we apply to the backup access times corresponding to different disks in each round. This reserves enough access time to ensure uninterrupted system operation for any particular failed disk. However, the reported measured time is taken when the disk 0 is assumed inaccessible (Figure 2). Overall, we believe that some limited discrepancy between predicted and measured access time leaves a reasonable cushion space for stable operation. This makes the system operation more robust, and guards it against nondeterministic factors, such as the sys-

<sup>3</sup>In previous work [3], we reported similar differences between the average reserved time and the access time measured when using the hardware disks of Table 2, instead of their simulated models.

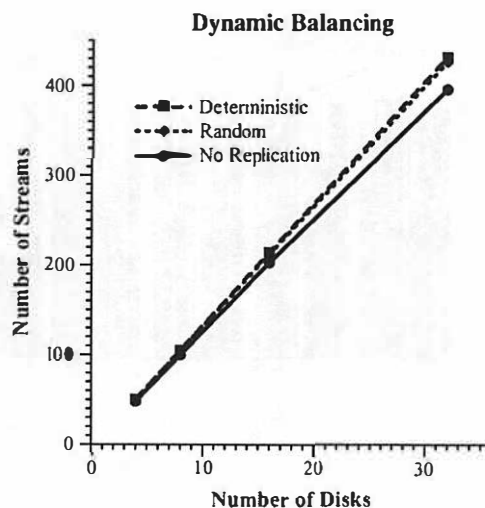


Figure 10: During normal operation, accessing the replica of the least loaded disk improves the throughput by about 5-10% with respect to the non-replicated case. The gain tends to increase as the disk array size increases.

tem bus contention due to network transfers, not included in the previous measurements.

### 7.3 Improving Load Balancing

With multiple data replicas available, better load balancing can be achieved by choosing the replica stored on the least loaded disk during admission control. In this case, we leverage data replication for improving the system throughput, rather than tolerating disk failures. We use the accumulated disk access time estimations in order to choose the least loaded disk. Making this choice based on actual measurement of the disk access load is not a feasible alternative, due to the round-based operation that prevents access load propagation from one round to the next.

From Figure 10, we see that, when this load balancing scheme is used, both replica placement policies can support 5-10% more streams than the non-replicated case. The difference between the two placement policies is statistically insignificant, however, since the gain from the dynamic replica access exceeds the improved load balancing of deterministic placement. Determining during admission control which disk will be used for each data access is a reasonable policy for removing hot spots in the disk array. However, under sequential workloads the different disks are equally utilized already, and only a limited additional performance benefit can be accrued with the above policy.

In Figure 11, we consider the case of declustering backup replicas across multiple disks using a fixed

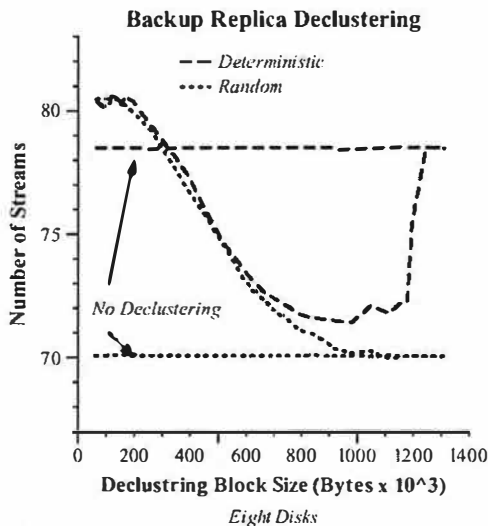


Figure 11: Each backup replica is divided into blocks of the specified size and distributed across multiple disks. There is a minor gain from applying backup replica declustering to deterministic placement, while the best throughput achieved with random placement approaches that of deterministic. The two horizontal lines correspond to the throughput achieved when no declustering is applied to the two replica placement policies, respectively.

block size  $B_d$ . This approach is expected to let the failed-disk load be more fairly shared among the surviving disks. With small block sizes, better load balancing leads into some limited throughput improvement. As the block size becomes larger, load balancing gets successively worse and throughput decreases, because declustering creates fragments with sizes increasingly different.

We also anticipate that, with larger block sizes, the number of disks accessed for each stream drops and the total head movement overhead becomes lower. On the other hand, our stride-based allocation scheme ensures that at most two head movements are required per stream on each disk regardless of how small the block size is. This keeps limited the negative effect of access overhead to throughput. Finally, we observe a threshold behavior around  $B_d = 1.2 \cdot 10^6$  bytes. This is the maximum amount of data retrieved in one round for each stream and originates from the bit-rate parameters used during encoding (Table 3). Effectively, beyond this point there is no declustering.

These observations are also verified by Figure 12, that shows the average difference between the access times of the most and least loaded disk in each round. Since the reserved access time of the most heavily loaded disk is typically 99% in each round, the plots in Figure 12 essentially indicate the ac-

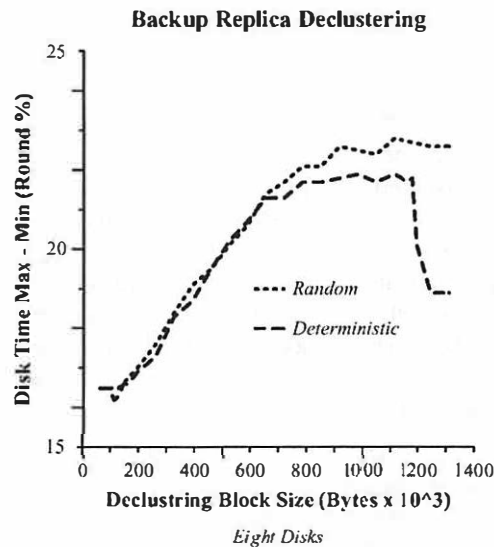


Figure 12: The average difference (expressed as round length percentage) in the reserved access times between the most and least loaded disk in each round varies according to the declustering block size. This difference can be interpreted as one measure of load imbalance within each round. From the shape, we see that it has a significant effect to the achieved throughput shown in Figure 11.

cess time requirements of the least loaded one. It is remarkable how the shape of the plots is reflected to those tracking the system throughput in Figure 11. We conclude that even the least loaded disk is expected to remain more than 80% utilized under deterministic replica placement with no declustering (equivalently, with declustering block size larger than  $1.2 \cdot 10^6$ ).

In summary, declustering is only worthwhile with small declustering block sizes, and its overall benefit is found to be limited in the media streaming case (less than 3% with eight disks). Moreover, the throughput of random replica placement never exceeds that of deterministic.

## 8 Discussion

We considered data replication and bandwidth allocation schemes that allow tolerating single disk failures in disk arrays storing variable bit-rate streams. When using simple schemes for reserving disk bandwidth, more than half of the maximum achievable throughput is wasted during normal (i.e. no failure) operation. Instead, using the minimum reservation scheme for accommodating a single disk failure results only in throughput reduction of less than 20% at disk array sizes sixteen or larger.

The minimum reservation scheme requires maintaining number of vectors equal to the square of the



number of disks. Each vector is accessed in a circular fashion and has minimum length equal to that of the longest stream expressed in numbers of rounds. When using large disk arrays, this might raise concerns regarding the computational and memory requirements involved. In practice, the reduction in unused bandwidth is diminishing as the number of disks increases beyond sixteen. Therefore, it makes sense to apply the data replication within disk groups of limited size, when the disk array size becomes larger. This keeps the bookkeeping overhead limited and preserves the scalability of our method when stream data are striped across large disk arrays.

In previous work, we found that striping data using fixed-size blocks achieves lower throughput than when using variable-grain striping [2]. The backup replicatedustering should not be confused with fixed-size block striping, since the primary data still use variable-grain striping. This maintains some benefit from multiplexing requests of different transfer sizes in each round, and absorbs correlations that otherwise would create maximum requirements much higher than the average.

Provisioning for VCR functionality is an important issue that we don't consider extensively in the present paper. In general, such flexibility would require deallocation of previously reserved resources, when a stream playback is suspended or stopped earlier than its normal termination. This can be done in a straightforward way, when accumulating disk access delays separately for primary and backup data replicas, as was already described above.

The techniques we presented here could be extended in straightforward ways for handling multiple disk failures. That would require storing multiple backup replicas, and making bandwidth reservations for more than one failed disk. In servers consisting of multiple nodes, failure of an entire node can also be handled gracefully, by keeping each disk of a node in a separate disk group and limiting the replication within each group. When a node fails, inaccessible data for each of its disks can be retrieved using replicas available on other disks of the corresponding groups [9, 15].

## 9 Related Work

Most of the previous work on disk array fault-tolerance has been done in the context of traditional file server and transaction processing workloads. Bitton and Gray show that mirrored disks can improve I/O performance in addition to providing enhanced reliability [8]. Hsiao and DeWitt describe chained dedus-

tering that replicates each database relation on two consecutive disks, while the workload is balanced across the system using a static load balancing algorithm [21]. Merchant and Yu propose using different stripe sizes for different data replicas [23]. Thus, system operation can be efficient with both small transaction requests and ad hoc queries on large parts of a relation.

In our previous work, we found the throughput measured with disk striping of variable bit-rate streams to increase linearly as a function of the number of disks [1, 2]. We also described several design decisions of our server prototype implementation [3]. The system throughput is further improved when the disk bandwidth requirements of individual streams are smoothed across different playback rounds [4], and high disk bandwidth utilization is achieved across both homogeneous and heterogeneous disks. System reliability is a crucial issue when building infrastructure for commercial services. Addressing this issue creates a strong case for storage of variable bit-rate streams, and makes the results of the present paper indispensable part of our previous published work.

The related work from media server research is mostly focused on fault-tolerance techniques when striping constant bit-rate streams [5, 6, 32]. Disks are grouped into clusters, and data blocks from separate disks in each cluster are combined with a parity block to form parity groups. The blocks of a parity group are considered to be retrieved and transmitted in one or multiple rounds, and the parity blocks are stored on data disks or dedicated parity disks. For improving overall efficiency, certain data blocks are not transmitted in a transition period following a disk failure.

Ozden et al. propose reading ahead the data blocks of an entire parity group prior to their transmission to the client [25]. When a data block cannot be accessed, it can be reconstructed using a parity block that is read instead. Alternatively, an entire parity group is retrieved each time a block cannot be accessed. Balanced incomplete block designs are used for constructing parity groups that keep the load of the disk array balanced [20, 25]. The dynamic reservation scheme that they introduce minimizes the extra bandwidth that has to be reserved on a disk for reconstructing failed-disk data blocks.

Gafsi and Biersack compare several performance measures of alternative data-mirroring and parity-based techniques for tolerating disk and node failures in distributed video servers [15]. When entire data blocks of one disk are replicated on different disks, half of the total bandwidth of each disk is reserved

for handling the disk failure case. The wasted throughput is critically reduced with the minimum reservation scheme that we propose here.

Tewari et al. study parity-based redundancy techniques for tolerating disk and node failures in clustered servers [30]. By distributing the parity blocks of an object on a random permutation of certain disks they can keep balanced the system load when a disk fails. Alternatively, Flynn and Tetzlaff replicate data blocks across non-intersecting permutations of disk groups [14]. Multiple available data blocks can be used for dynamic balancing of disk bandwidth utilization across different devices. Instead, Birk examines selectively accessing parity blocks of video streams for better balancing the system load across multiple disks [7].

For failures in video servers supporting variable bit-rate streams, Shenoy and Vin apply lossy data recovery techniques that rely on the inherent redundancy in video streams rather than error-correcting codes. Alternatively, they propose taking advantage of the sequential block accesses during playback and reconstructing missing data from surrounding available blocks, at the cost of an initial playback latency, or temporary disruption when a failure occurs [29].

Bolosky et al. deduster the block replicas of one disk across  $d$  other disks. In case of disk failure, the extra bandwidth required for retrieving the data of the failed disk is shared among the  $d$  other disks [9]. In later work, they also consider providing fault-tolerant support for multiple streams with different bit rates [10]. In our experience, declustering does not add significant improvement with respect to the case of replicating the data blocks of one disk in their entirety on different disks.

Mourad describes the doubly-striped disk mirroring technique that distributes replica blocks of one disk round-robin across the rest of the disks [24]. The system load is equally distributed across the surviving disks in case of a disk failure. The deterministic replica placement that we describe extends doubly-striped mirroring for handling variable bit-rate streams and the reduced device utilization that they potentially introduce.

Santos et al. compare disk striping against data replication on randomly chosen disks [27]. Using constant bit-rate streams, they conclude that random replication can outperform disk striping with no replication. In our comparison using variable bit-rate streams instead, we found an advantage of deterministic replication over random replication that

diminishes as the number of disks increases.

## 10 Conclusions

We studied issues related to data replication of variable bit-rate streams striped across multiple disks for improving system reliability and performance. We introduced the *minimum reservation scheme* that minimized the wasted throughput required for keeping accepted playbacks uninterrupted during a disk failure. At moderate disk array sizes, the throughput is less than 20% lower than what is achieved with no replication. Deterministic placement of backup data is found to achieve better performance than random placement across the different disks, although the advantage becomes insignificant as the number of disks increases. Retrieving the data replica of each stream stored on the least loaded disk adds an improvement of no more than 10% with respect to the non-replicated case. Finally, declustering the backup replicas across multiple disks does not seem to considerably improve the performance achieved with deterministic replica placement.

## References

- [1] Anastasiadis, S. V. *Supporting Variable Bit-Rate Streams in a Scalable Continuous Media Server*. PhD thesis, Department of Computer Science, University of Toronto, June 2001.
- [2] Anastasiadis, S. V., Sevcik, K. C., and Stumm, M. Disk Striping Scalability in the Exedra Media Server. In *ACM/SPIE Multimedia Computing and Networking Conference* (San Jose, CA, Jan. 2001), pp. 175–189.
- [3] Anastasiadis, S. V., Sevcik, K. C., and Stumm, M. Modular and Efficient Resource Management in the Exedra Media Server. In *USENIX Symposium on Internet Technologies and Systems* (San Francisco, CA, Mar. 2001), pp. 25–36.
- [4] Anastasiadis, S. V., Sevcik, K. C., and Stumm, M. Server-Based Smoothing of Variable Bit-Rate Streams. In *ACM Multimedia Conference* (Ottawa, ON, Oct. 2001), pp. 147–158.
- [5] Berson, S., Ghandeharizadeh, S., Muntz, R., and Ju, X. Staggered Striping in Multimedia Information Systems. In *ACM SIGMOD Conference* (Minneapolis, MN, May 1994), pp. 79–90.
- [6] Berson, S., Gohrbachik, L., and Muntz, R. R. Fault Tolerant Design of Multimedia Servers. In *ACM SIGMOD Conference* (San Jose, CA, May 1995), pp. 364–375.
- [7] Birk, Y. Random RAIDs with Selective Exploitation of Redundancy for High Performance Video Servers. In *International Workshop on Network*

- and Operating System Support for Digital Audio and Video (Zushi, Japan, May 1997), pp. 13-23.
- [8] Bitton, D., and Gray, J. Disk Shadowing. In *Very Large Data Base Conference* (Los Angeles, CA, Aug. 1988), pp. 331-338.
  - [9] Bolosky, W. J., Barrera, J. S., Draves, R. P., Fitzgerald, R. P., Gibson, G. A., Jones, M. B., Levi, S. P., Myhrvold, N. P., and Rashid, R. F. The Tiger Video Fileserver. In *International Workshop on Network and Operating System Support for Digital Audio and Video* (Zushi, Japan, Apr. 1996), pp. 97-104.
  - [10] Bolosky, W. J., Fitzgerald, R. P., and Douceur, J. R. Distributed Schedule Management in the Tiger Video Fileserver. In *ACM Symp. Operating Systems Principles* (Saint-Malo, France, Oct. 1997), pp. 212-223.
  - [11] Chang, E., and Zakhor, A. Cost Analyses for VBR Video Servers. *IEEE Multimedia* (Winter 1996), 56-71.
  - [12] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys* **26**, 2 (June 1994), 145-185.
  - [13] General Information on Cable TV and its regulation. Fact Sheet, Federal Communications Commission, June 2000. <http://www.fcc.gov/csb/facts/csgen.html>.
  - [14] Flynn, R., and Tetzlaff, W. Disk Striping and Block Replication Algorithms for Video File Servers. In *IEEE International Conference on Multimedia Computing and Systems* (Hiroshima, Japan, June 1996), pp. 590-597.
  - [15] Gafsi, J., and Biersack, E. W. Modeling and Performance Comparison of Reliability Strategies for Distributed Video Servers. *IEEE Transactions on Parallel and Distributed Systems* **11**, 4 (Apr. 2000), 412-430.
  - [16] Ganger, G. R., Worthington, B. L., and Patt, Y. N. The DiskSim Simulation Environment: Version 2.0 Reference Manual. Tech. Rep. CSE-TR-358-98, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan, Dec. 1999.
  - [17] Gray, J., and Shenoy, P. Rules of Thumb in Data Engineering. In *IEEE International Conference on Data Engineering* (San Diego, CA, Feb. 2000), pp. 3-10.
  - [18] Gringeri, S., Shuaib, K., Egorov, R., Lewis, A., Khasnabish, B., and Basch, B. Traffic Shaping, Bandwidth Allocation, and Quality Assessment for MPEG Video Distribution over Broadband Networks. *IEEE Network*, 6 (Nov/Dec 1998), 94-107.
  - [19] Haskin, R. L., and Schmuck, F. B. The Tiger Shark File System. In *IEEE COMPCON* (Feb. 1996), pp. 226-231.
  - [20] Holland, M., and Gibson, G. A. Parity Declustering for Continuous Operation in Redundant Disk Arrays. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, Oct. 1991), pp. 23-35.
  - [21] Hsiao, H.-I., and DeWitt, D. J. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *IEEE International Conference on Data Engineering* (Los Angeles, CA, Feb. 1990), pp. 456-465.
  - [22] Lakshman, T. V., Ortega, A., and Reibman, A. R. VBR Video: Tradeoffs and Potentials. *Proceedings of the IEEE* **86**, 5 (May 1998), 952-973.
  - [23] Merchant, A., and Yu, P. S. Analytic Modeling and Comparisons of Striping Strategies for Replicated Disk Arrays. *IEEE Transactions on Computers* **44**, 3 (Mar. 1995).
  - [24] Mourad, A. Doubly-Striped Disk Mirroring: Reliable Storage for Video Servers. *Multimedia Tools and Applications* **2** (May 1996), 273-297.
  - [25] Ozden, B., Rastogi, R., Shenoy, P., and Silberschatz, A. Fault-tolerant Architectures for Continuous Media Servers. In *ACM SIGMOD Conference* (Montreal, Canada, June 1996), pp. 79-90.
  - [26] Patterson, D. A., Gibson, G., and Katz, R. H. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM SIGMOD Conference* (Chicago, IL, June 1988), pp. 109-116.
  - [27] Santos, J. R., Muntz, R. R., and Ribeiro-Neto, B. Comparing Random Data Allocation and Data Striping in Multimedia Servers. In *ACM SIGMETRICS Conference* (Santa Clara, CA, June 2000).
  - [28] *Cheetah 36XL Product Manual*. Seagate Technology, Feb. 2001. <http://www.seagate.com/cda/products/discsales/enterprise/family/0,1130,318,00.html>.
  - [29] Shenoy, P. J., and Vuu, H. M. Failure Recovery Algorithms for Multimedia Servers. *ACM Multimedia Systems Journal* **8**, 1 (Jan. 2000), 1-19.
  - [30] Tewari, R., Dias, D. M., Mukherjee, R., and Vin, H. M. High Availability in Clustered Multimedia Servers. In *IEEE International Conference on Data Engineering* (New Orleans, LA, Feb. 1996), pp. 336-342.
  - [31] Thekkath, C. A., Wilkes, J., and Lazowska, E. D. Techniques for File System Simulation. *Software-Practice and Experience* **24**, 11 (Nov. 1994), 981-999.
  - [32] Tobagi, F. A., Pang, J., Baird, R., and Gang, M. Streaming RAID - A Disk Array Management System for Video Files. In *ACM Multimedia Conference* (Anaheim, CA, Aug. 1993), pp. 393-400.

# Simple and General Statistical Profiling with PCT

Charles Blake

Laboratory for Computer Science  
Massachusetts Institute of Technology  
cb@mit.edu

Steve Bauer

Laboratory for Computer Science  
Massachusetts Institute of Technology  
bauer@mit.edu

## Abstract

The Profile Collection Toolkit (PCT) provides a novel generalized CPU profiling facility. PCT enables arbitrarily late profiling activation and arbitrarily early report generation. PCT usually requires no re-compilation, re-linking, or even re-starting of programs. Profiling reports gracefully degrade with available debugging data.

PCT uses its debugger controller, `dbctl`, to drive a debugger's control over a process. `dbctl` has a configuration language that allows users to specify context-specific debugger commands. These commands can sample general program state, such as call stacks and function parameters.

For systems or situations with poor debugger support, PCT provides several other portable and flexible collection methods. PCT can track most program code, including code in shared libraries and late-loaded shared objects. On Linux, PCT can seamlessly merge kernel CPU time profiles with user-level CPU profiles to create whole system reports.

## 1 Introduction

Profiling is the art and science of understanding program performance. There are two main families of profiling techniques, automatic code instrumentation and statistical sampling. Code instrumentation approaches use a high-level language compiler or linker to incorporate new instructions into object file outputs. These instructions count how many times various parts of a program get executed. Some instrumentation systems [12] count function activations while others [1, 21] count more fine-grained control flow transitions. Sampling approaches mo-

mentarily suspend programs to sample execution state, such as the value of the program counter. How frequently certain locations occur during an execution estimates the relative fraction of time incurred by those parts of the program.

PCT is a sampling-based profiling system that shows a new way to construct effective performance investigation tools. PCT demonstrates that the same tools programmers are familiar with for answering questions about correctness can be used for effective performance analysis. The philosophy of PCT is that profiling is a particular type of debugging and that the same preparations should be adequate. The focus of PCT is CPU-time profiling rather than real-time profiling, though, in principle, sampling may be applied to either.

PCT is also flexible and easy to use. Enabling profile collection rarely requires re-compiling, re-linking, or even re-starting a program. In its simplest usage, adding a one word prefix to the command-line can activate collection over entire process subtrees and emit a basic analysis report at the end. PCT can track CPU time spent in the main program text, shared libraries, late-loaded dynamic objects and in kernel code on Linux. PCT works with a variety of programming languages.

A novel aspect of PCT is that it allows sampling semantically rich data such as function call stacks, function parameters, local or global variables, CPU registers, or other execution context. This rich data collection capability is achieved via a debugger-controller program, `dbctl`. Using debuggers to probe program state allows PCT to sample a wide variety of values. Statistical patterns in these values may explain program performance. For example, statistically typical values of a function's parameters may explain *why* a program spends a lot of time in that function.

Additionally, `dbctl` can drive parallel non-

Feature	Description
Causally Informative	Maximize ability to explain performance characteristics
Extensible	Sample many kinds of user-defined program state
Late-binding	Defer as long as possible the decision of whether to profile
Early-reporting	Make profile reports available as soon as possible
Non-invasive	Require no extra program build steps or copies of objects
Low-overhead	Minimize extra program run time
Portable	Support informative profiles on any OS and CPU
Robust	Do not rely on program correctness, in particular clean exits
Tolerant	Report quality should gracefully degrade with worse system support, poorer profile data, and less rich debugging data in object files.
Exhaustive	Track as much relevant CPU activity as possible
Multilingual	Support different programming languages, multi-language environments

Table 1: Desirable profiling system features.

interactive debugging sessions. As the original process creates children, dbctl can spawn off new debugger instances, reliably attaching them to those children. Using a debugger-controller allows a *portable* implementation of process subtree execution tracing tools, such as function call tracers.

The functionality of PCT gracefully degrades with the available support in the system and the executables of interest. Debugging data or symbol tables are needed for highly meaningful reports. Nevertheless even stripped binaries allow some analysis. For example, one can track the usage of dynamic library functions or emit annotated disassembly. In concert with instrumentation-based basic block-level profiling such as gcov, PCT can even estimate CPU cycles per instruction. Sampled data can be windowed in time to isolate different CPU intensive periods of a program's execution. The various report formats are available through a set of composable primitive programs and shell pipelines.

Profile reports may be generated at any time, even prior to program termination and several times over the life of one process. Several granularities are available for data aggregation and report formats. Depending on the debugging data available in executables, users can select how to display program locations. This may be at the level of individual instructions, line numbers, functions, source files, or even whole object files or libraries.

The organization of this paper is as follows. Section 2 discusses our design objectives. To make PCT's capabilities more concrete, Section 3 shows a few examples. Section 4 then elaborates upon PCT's implementation of data collection. Section 5 details

report generation strategies. Section 6 evaluates the overhead and accuracy of the toolkit. Section 7 discusses some other approaches to profiling. Section 8 describes how to obtain the software. Finally, Section 9 concludes.

## 2 Design Objectives

The design goals of PCT were driven by user needs and the inadequacies or inaccessibility of prior systems. Table 1 highlights these objectives. The following section argues for the importance of each in turn, and the approach of PCT in general.

Programmers use profiling systems to understand what causes performance characteristics. E.g., if certain functions dominate an execution, then a profile should tell us why those calls are made, and why they might be slow. If functions are called with arguments implying quite different "job sizes", then a profile should be able to capture this for analysis. Exactly how *causally informative* profiling can or should be is an open issue. More information is better up until some point where overhead and analysis tractability concerns become a problem. Programmers currently have far more *a priori* knowledge about what to look for than any automatic system can hope to have. A practical answer is an *extensible collection* system that lets users decide what program variables are most relevant to subsequent performance analysis.

Performance problems often arise only on inputs by end users unanticipated by the programmer or in very late stages of testing. These issues are thus



discovered at the worst possible time for rebuilding a program and all its dependencies. Long running programs such as system services often have phased behavior. That is, sections of the program with quite distinct performance characteristics execute over various windows in time. Profiles over entire program executions can introduce unwanted averaging over this phased behavior, making results more difficult to interpret. A direct and flexible way to address this problem is to allow *late-binding*. Ideally, activating and deactivating profile collection should be possible at *any* stage in the life cycle of a program. As an immediate correspondent, *early-reporting* is also desirable so that long running programs with highly active phases do not need to terminate before a profile can be examined. Together, these let programmers apply whatever knowledge they have about phased behavior.

Classic instrumentation techniques raise a number of administrative, theoretical, and practical issues. Instrumentation usually requires extra steps to build two versions of executables and libraries, ordinary and instrumented. It is often problematic to require recompilation of all objects in all libraries or to require commercial vendors to provide multiple versions of their libraries. Providing multiple library versions can be a burden even on the various contemporary open source platforms. For instance, profiling instrumented libraries in `/usr/lib` on open source distributions are scarce or entirely absent. Also, it is possible to instrument code long after linking it. For example, binary rewriting techniques along the lines of Pixie [23] or Quantify [14] allow this. Completely dynamic instrumentation is also possible.[18]

Nonetheless, instrumentation, at whatever time, raises several issues. Instrumented code really is not the same as the original code. Subtle microarchitectural effects can make it hard to understand the overhead of new instructions. Beyond theoretical accuracy issues, there is also a more practical concern in that getting the instrumentation correct is a challenging problem in itself. Profiling instrumentation can interact badly with “new” compiler features, optimization strategies, or uncommon language usage patterns. In the worst case, which is all too frequent, the produced executable may not even run correctly. Finally, with the possible exception of fully dynamic instrumentation, this strategy is inherently less extensible. Only *a priori* data types can be extracted, and this is usually limited to simple counts of executions to avoid re-implementing a good deal of

compiler technology.

While instrumentation has the virtue of precision, the above considerations suggest that we should go as far as possible with systems that are *non-invasive* to the stream of instructions the CPU encounters. In essence, this implies a sampling-based approach. Sampling also has the virtue of incurring tunably *low overhead*.

Performance problems often arise only when programs are used in very different environments from where they were developed. Platform-specific profiling packages can be more efficient and occasionally more capable. However, they do not help if performance problems cannot be reproduced on supported platforms or environments. Programmers also have a rational resistance to learning and relying upon multiple, disparate system-specific tools and interfaces. Therefore, a more *portable* system is more valuable.

Portability concerns also suggest a sampling approach. Any preemptive multi-tasking OS already suspends and resumes programs as a matter of course. The only missing pieces for profile collection are a means to suspend frequently and a mechanism to inspect the state of a program. Reading a program’s state is inherently simpler than re-writing its code. Thus, sampling is typically no more intrusive than ordinary preemption and requires simpler, less specialized system support than automatic instrumentation.

Some past profiling systems have used in-core buffers that are written to disk in `atexit()` handlers at the end of a clean program shutdown. A system should not mandate clean termination in order to diagnose performance problems. One *phase* of a program may warrant performance investigation even if other phases are buggy. Inputs needed to trigger performance pathologies may also instigate incorrect behavior. Conversely, performance pathologies can easily trigger failure modes not ordinarily encountered. Thus, profile collection should ideally be *robust* against program failure. Many existing implementations could be adapted to be more cautious in this regard.

Bottleneck code can potentially hide anywhere in a program. Restricting profiling coverage to only code compiled or linked into the address space in certain ways leads to many “holes” in the accounting of where execution time was spent. The more *exhaus-*

*tive* code coverage is, the more likely a profile will unravel performance mysteries.

Mixed programming language systems have become pervasive in modern software development environments. While C and C++ are a canonical example, it can be the case that a wider range of languages, e.g. FORTRAN, Ada are supported within one program. If unified source-level debugging exists for these *multilingual* environments then source-level profiling should also be supported. A profiling system wedded to a particular programming language or code generation system is too inflexible.

PCT is the first profiling system known to the authors to possess *all* of these properties simultaneously. Extensible and informative data collection is achieved through the ability of source-level debuggers to compute arbitrary expressions and do detailed investigation of program state. PCT is non-invasive and low-overhead since sampling does little more than what the OS ordinarily does during task switching. The sampling rate can be changed to trade-off overhead with accuracy. Late-binding is achieved by delaying activation of sampling code or having it instigated by an entirely different process, namely the debugger. Portability derives from relying only on old, well-propagated system facilities dating back to the mid-1980s. Robustness ensues from the earliest possible commitment of data to the OS buffer cache, which is closely related to producing reports as soon as any data has been collected. The system is as multi-lingual as the executable linking environment allows. PCT is as exhaustive as the debugger, OS, and build environment allows. The very small report generation modules enable tolerating various levels of debugging data, customizing reports, and porting PCT to a new platform.

PCT is also a small system. The code for PCT is only 3,500 commented lines of C code and 300 lines of shell scripts. This compact delivery of functionality is possible only because PCT greatly leverages common system facilities.

### 3 Examples

On many systems getting a quick profile is as simple as: `profile myprogram args...`

More concretely:

```
$ profile ./fingerprint /bin | head
13.9%  /u/cblake/hashfp/binPoly64.C:101
 7.6%  /u/cblake/hashfp/binPoly64.C:86
 7.3%  /lib/libc-2.1.2.so:getc
 5.3%  /u/cblake/hashfp/fingerprint.C:112
 4.3%  /u/cblake/hashfp/binPoly64.C:80
 4.2%  /u/cblake/hashfp/binPoly64.C:70
 4.0%  /u/cblake/hashfp/binPoly64.C:96
 3.7%  /u/cblake/hashfp/binPoly64.C:102
 3.0%  /u/cblake/hashfp/fingerprint.C:116
 2.8%  /u/cblake/hashfp/fingerprint.C:104
```

By default line numbers are used for source coordinates. If only symbols are available they are used. Finally, raw *objectfile:address* pairs are printed when there is no debugging data at all.

Profiling mixed kernel and user code on Linux is similar. Below is a quick profile of the disk usage utility which recurses down a directory tree summing up file allocations:<sup>1</sup>

```
$ profile -k du -s /disk/pa0 | head
30.4%  /usr/src/linux/vmlinux:iget4
12.4%  /usr/src/linux/vmlinux:ext2_find_entry
 5.4%  /usr/src/linux/vmlinux:try_to_free_inodes
 3.5%  /usr/src/linux/vmlinux:ext2_read_inode
 3.3%  /usr/src/linux/vmlinux:unplug_device
 2.4%  /usr/src/linux/vmlinux:lookup_dentry
 2.0%  /usr/src/linux/vmlinux:system_call
 1.8%  /usr/src/linux/vmlinux:getblk
 1.0%  /lib/libc-2.1.2.so:open
 0.9%  /lib/libc-2.1.2.so:__lxstat64
```

Note the call hierarchy in the following program:

```
int worker(unsigned n) { while (n--) /**/ ; }
int dispatch_1(unsigned a) { worker(a); }
int dispatch_2(unsigned b) { worker(b); }

int main(int ac, char **av) {
    dispatch_1(10000000);
    dispatch_2(20000000);
    return 0;
}
```

Before doing any profiling it is obvious that essentially all run time is in the function `worker()`. There are two paths to this function, as shown clearly via the debugger-based hierarchical profile:

```
$ profile -gdb -l3 hier-test
67.6%  worker  <-  dispatch_2  <-  main
32.4%  worker  <-  dispatch_1  <-  main
```

<sup>1</sup>Directory and i-node data was pre-read to make the results reflect CPU time spent in the 2.2 kernel.

Finally, consider sampling more semantically rich data. In general this requires amending a 10 line `dbctl` script similar to the following:

```

1 EXEC() ".*" {
2 #include "gdbprof_prologue.dbctl"
3 PAT_GROUP(default) {
4   PAT(1) "signal=\SIGVTALRM\" | OUT("pc") {
5     "backtrace 4" | OUT("stack");
6     # OTHER DEBUGGER EXPRESSIONS
7     "continue";
8   }
9 #include "gdbprof_epilogue.dbctl"
10 }
11 }

```

A full description of the pattern-driven state machine language is beyond the scope of this paper. The main `EXEC` pattern on line 1 restricts which executables the entire rule applies to. Lines 4, 5, and 6 simply capture the `pc` in the output file `"pc"`, and four levels of stack backtrace in the output file `"stack"`. Adding more debugger commands and data files is just a matter of adding a line to the `dbctl` script. Depending on the expressions sampled, various post-processing steps may be needed.

We provide a simpler interface for the common case of sampling scalar numbers. Below shows how to sample values of `n`, inside the function `worker()` where it is meaningful.

```

$ profile -gdb \
  -expr 'hier-test@worker@n' 'int-avg' \
  hier-test
8.38e+06

```

The `-expr` option takes two arguments – a context-specific expression to generate data in the debugger, and a program to format the collected data. The context specific expression is an '@' separated tuple of strings: a program pattern, a function pattern, a debugger expression.

## 4 Data Collection

The general implementation philosophy of PCT is to support a full set of options for every aspect of profiling. This minimizes the chance that some system limitation will prevent any profiling outright and enables "best effort" profiling. Small, composable

primitives also ease tailoring PCT behavior. Basic users generally use several generic driver scripts, while more advanced users create their own tailored script wrappers.

### 4.1 Activating Sampling Code

PCT has three basic collection strategies: debuggers, timer signal handlers, and `profil()`. [5] The first never requires re-starting or re-linking a program, but can have substantial real-time overhead. The latter two are fall-back, library-based strategies which can be used when low overhead is preferable or when debugger support is inadequate. The library-based samplers are, however, less portable. They require linker support for C++-style global initializers and also require either a dynamic library pre-loading facility or manual re-linking. Dynamic pre-loading is commonly available with modern dynamic linkers [3], though not all programs are dynamically linked. In the worst case, if C++-style linking is unavailable, programmers can manually invoke the initializer inside their `main()` routine. We now examine these samplers in more detail.

The oldest portable profiling primitive is `profil()`. This system call directs kernel-resident code to accumulate a histogram of program counter locations in a user-provided buffer. While the call interface does potentially allow multiple executable regions, the authors know of no operating systems that can activate more than one region at a time. To ensure robustness, PCT allocates the userspace buffer as an `mmap()`-ed file. This also allows the profile to be accessible at any time to other processes, such as report generators. The `profil()` call to activate kernel-driven collection can be done with either a dynamically pre-loaded or statically linked-in library.

A source-level debugger affords a more general sampling activation strategy. The debugger uses the `ptrace()` facility and catches all signals delivered to the process, including virtual timer alarms. `ptrace()` can be used to attach and detach from processes at any time and any number of times over the lifetime of a process. PCT uses a debugger-controller program to implement this procedure portably.

The PCT debugger controller drives the debugger which in turn controls the process via `ptrace()`. The debugger calls the POSIX `setitimer()` sys-

tem call in the context of the target process. This installs virtual time interval timers for the target process. Once these timers are installed, the kernel will deliver **VTALRM** signals periodically to the target process. At each signal delivery, control will be transferred to the debugger. At this point the debugger driver issues whatever debugger commands are necessary to collect informative data and then continue program execution. For example a **backtrace** or **where** command typically produces a sample of function call stack data. The real time of the sample can also be recorded. Section 4.3 discusses the details of debugger control.

When library code can be used, a **pre-main()** initializer sets up interval timers, signal handling, and data files. **gcc-specific**, **C++**, or **system-dependent** library section techniques can be used to install the library initializer. Library code may be statically or dynamically linked, or preloaded for dynamically-linked executables via the **\$LD\_PRELOAD** environment variable.

PCT collection behavior is controlled through the **\$PCT** environment variable. It controls options such as output directories, histogram granularity, data format, and so on. It also provides a convenient switch for whether profiling happens at all. **\$PCT** and **\$LD\_PRELOAD** can both be inherited across **fork()** and **exec()**. This conveniently enables profile collection activation on whole process subtrees.

## 4.2 Collecting Data

### 4.2.1 Types of Code

The debugger collector supports tracing whatever code the debugger can recognize. All debuggers handle the main program text. Most modern debuggers, e.g. **gdb**, can debug code in shared libraries and late-loaded object files on most operating systems.

PCT library-based collectors have more specific restrictions. They can collect data on several kinds of code:

- One contiguous region – usually the main program text. This is the oldest style of profiling and works in almost any OS and scenario.
- Shared libraries. On Linux the instantaneous bindings of virtual memory regions to files are

exposed. Reading **/proc/PID/maps** reveals executable regions and corresponding object files.

On most BSD OSes **ldd** reports load addresses of shared libraries. These addresses may be cached in files similar to **/proc/PID/maps** and read in by the global initializer.

- Late-loaded (e.g. **dlopen()**) code: On Linux whenever a PC cannot be mapped to a known memory region, the signal handler re-scans **/proc/PID/maps** to attempt to discover new regions. If it succeeds, the region table is updated and the counts processed. When an object file for the PC cannot be found, further re-scanning is inhibited to suppress repetitive failed searches.
- Kernel code: Linux provides a **/proc/profile** buffer for the main text of the kernel. Currently, Linux does not support profiling loadable kernel modules.

As mentioned in Section 4, **profil()**-based collection is generally only available for a single contiguous region of address space. These other types of code are all supported by the more general library-based collector. Profiling kernel modules could be added to Linux or other OS's using the same techniques that PCT uses for managing shared libraries and late-loaded code.

### 4.2.2 Types of Profiling Data

Collection methods based on **profil()** or **/proc/profile** afford little choice as to the type or format of data collected. Other sampling methods, such as **\$LD\_PRELOAD** and debuggers allow collecting a variety of data. This flexibility creates choices as to *what* data is collected for later analysis, how and where it is stored.

PCT provides several data storage formats. The specific profiling situation will usually determine which is best. The choices are:

- Debugger output files: a different log file is used to save the output of each user-specified sampling expression.
- Histogram file: stores frequency counts for various code regions. This guarantees bounded space, but cannot window data.

- Sample-ordered log file: allows simple time-windowing of collection events, post facto histograms, but can grow in size indefinitely.
- Circular log file: This is similar to the sample-ordering except that the user bounds the size, which effectively saves only the last  $N$  samples.

### 4.3 Controlling Debuggers

The PCT toolkit includes a controller program `dbctl` for driving debugger tools such as `gdb`. The controller program is a state machine described by user specified files. A transition in the state diagram occurs when the controller recognizes a regular expression in the output of the debugger. For each transition, there is a set of debugger commands to issue as well as a series of controller actions. The debugger commands can be any command appropriate for the debugger tool being controlled. For example, controller actions might include logging debugger output, spawning new debuggers to attach to child processes, and capturing specified debugger outputs in internal controller variables.

In the case of profiling, the `dbctl` tool sets up the interval timers in the processes to be profiled. When the timers expire a signal is raised which transfers control to the debugger and generates output indicating the context of suspension. The controller recognizes various process contexts and issues context-specific instructions. These can include writing out the call stack, local variables and function arguments, or any arbitrary debugger expression.

Profiling is not the only application of `dbctl`. The tool can also be used to implement a portable `strace` [7] or `ltrace` [11] facility. While many debuggers have function call tracing capabilities, tracing an entire *process tree* is more challenging.

The UNIX `ptrace()` mechanism has traditionally had rather weak support for following both a parent and child process across a `fork()`. Typically, there is a constraint of one-to-one binding between a traced process and a tracing process. After a `fork()` only one of the potentially traced processes can remain under external control. The other is released to be scheduled by the OS. Breakpoints left in a untraced process cause a `SIGTRAP` that causes the process to die since it has no debugger to catch the signal on its behalf. Therefore a debugger arranges things so that breakpoints are disabled across a `fork()` for

one of the two processes.

For `dbctl` this means that if we arrange to follow the parent, then a `fork()`ed child could “run away” from the controller, possibly `fork()`-ing grandchildren before `dbctl` can attach a new debugger to it. Similarly, if we arrange to follow the child, the parent could run away forking other children before a new debugger can be attached. Ideally, kernels would provide a standard interface to “fork and suspend” `ptrace()`d processes. Lacking a natural interface to avoid this race condition, PCT developed an interesting work around. Our `fork()`-following protocol guarantees that no child process is ever lost and that breakpoints can be re-enabled immediately after the call to `fork()`.

The protocol works as follows. First, we set a breakpoint at all `fork()` calls to catch the spawning of children. When the `fork()` breakpoint is hit, the debugger disables all breakpoints so that the untraced process will not get spurious `SIGTRAP` signals. It then installs `pause()` as a signal handler for `SIGTRAP`. Finally, it sets a breakpoint for the instruction following the call to `fork()`.

The parent remains `ptrace()`d all along, and immediately traps to the debugger because of the `fork()`-return breakpoint. All normal breakpoints are re-enabled. The child process id can be found on the stack as the return value from the `fork()`. `dbctl` uses this pid to attach a new instance of the debugger to the child process.

Concurrently, the `fork()`-return breakpoint causes a `SIGTRAP` to be delivered to the child. Since it no longer has a tracing process, the process’ own signal handler is invoked. In this case that function is `pause`, a system call which simply waits until some signal is delivered. When a newly spawned debugger successfully attaches to the process it interrupts the ongoing pause. The procedure of attaching a new debugger is now complete. `dbctl` then re-establishes any necessary breakpoints and so on in both processes and lets them run again.

### 4.4 Limitations

Library-based histogram collectors face a problem with `fork()`d processes/threads which truly run in parallel (i.e. on multiple CPU systems). The parallel processes can potentially overwrite each other’s counter updates. The result is a missed counter in-



crement with the last writer winning the counter bump. This is rare and is probably not an issue in practice. In any event, one can assess the number of lost counts by the total scheduling time given and the total counts collected. If there is a major discrepancy one can switch to log-file profile collection which does not share this problem.

Hierarchical samples are currently only supported with the debugger collector. The code to walk back a stack frame is conceivably simple enough for some CPUs to embed directly into our signal handler library. This could drastically reduce overhead at the cost of sacrificing some CPU portability and probably some language neutrality.

Our debugger controller requires that executables either be dynamically linked to the C library, or if statically linked contain a few critical symbols, such as `pause`, that may not be strictly required to be present. Of course the debugger can also do very little with executables stripped of all symbol data.

Sampling rates are limited by maximum `VTALRM` delivery rates. These typically range from 1 to 10 *ms*. Some systems allow increasing this rate. Depending on the richness of collected data, it may not be desirable to increase this rate, as that would entail more real-time overhead.

## 5 Data Analysis

PCT data collection strategies produce files with quite different information. Designing one monolithic way of reducing this data to programmer interpretable relationships is hard. Instead PCT provides a toolkit of data aggregation and transformation programs. These can be easily composed via UNIX shell pipelines. Their usage is simple enough that users can tailor simple scripts toward individual circumstances and preferences.

### 5.1 Source Coordinate Resolution

Debuggers emit high-level source coordinates as a matter of course. They are constrained by how much debugging data was been compiled into the executables and libraries being used. If these object files have been compiled with the full complement of debugging data then source and line number-level,

function-level, and address-level coordinates are all available. Usually, all are present in the textual debugger output PCT records. This mode of data collection then yields a lot of choices. PCT lets users select the coordinates to be used in reports.

On the other hand, library-based collectors do not resolve program counter addresses to source coordinates while the program is running. Instead they record only program counter addresses and defer higher-level coordinate translation to report generation-time. These addresses are saved in compact binary data formats that keep logs small and minimize IO overhead. There is usually one binary data file for each independently mapped region of program address space. Embedded within these files are the path names and in-memory offsets of the memory-mapped object files. This provides the key information for deferred translation to understand how addresses in memory correspond to addresses in the object files.

Translation of program addresses to more meaningful source coordinates can be awkward. Object file formats vary substantially. The GNU binary file descriptor library gives some relief, allowing the writing of programs which directly access debugging data and symbol tables. This library may be unavailable, out of date, or not support the necessary object file formats. As the examples in Section 3 show, PCT makes a best-effort attempts to translate coordinates.

At the least, if executable files retain their symbol table, the system `nm` program or the debugger can interpret it. If there is no debugger, the *GNU binutils* package provides a convenient `addr2line` program which can map PCs to file:line source coordinates. If there is a debugger installed, then a debugger script can operate just as `addr2line` in the restricted capacity of address translation. If there is no debugging data at all, as for stripped binaries, then a disassembly procedure is always an option. Indeed, for instruction-level optimization, it may even be desired to produce count-annotated disassembly files as reports. Of course, assembly-level expertise is required to interpret such reports.

PCT provides a printing program `pct-pr` which bridges the gap between PCT binary data files and programs which affect address translation. `pct-pr` assumes a simple and convenient protocol for shell pipeline syntax. Translators are run as co-processes to `pct-pr`. I.e., programs read a series of PCs on

their standard input and emit corresponding source coordinates to their standard output.

This co-process setup allows fine-tuning of the protocol with `pct-pr` command-line arguments. For example, `printf()`-style format strings allow tailoring the object file PC stream to input requirements, and also allow customizing the output stream. Users can stamp PCT binary data files with particular PC translation requirements, or simply describe which translators to use on the `pct-pr` command line.

PCT also provides a program `addr2nm` to translate PCs using only the system `nm` and symbol tables in executables. This program first dumps `nm` output into a cache directory if it is not already there. Once this file exists, `addr2nm` does an approximate binary search on each inbound PC, discovering the symbol with the greatest lower address.

## 5.2 Data Aggregation

One often wants to aggregate profile data over various uses of the same programs or libraries. The canonical example is combining many runs of short-lived programs. In the context of profiling a process tree, one may want to examine many distinct processes as one aggregate set of counts. For example, a `libc` developer might be interested in all the usages of some particular function, e.g. `printf()`, throughout a process tree. PCT supports these various styles of aggregation via simple filename conventions and traditional UNIX filename patterns. A user can select collections of data files by common filename substrings such as the name of the object files of interest. For example, `pct-pr /tmp/pct/ct/myprogram.*/libc*` would generate source coordinates for all samples of the `libc` code used by `myprogram`.

The output of source coordinate translation for each file in a collection is a simple pair of sample counts and labels for that location in the program. The granularity of these labels, e.g. function or source:linenumber, will determine the notion of similarity for later tabulation. This stream can be sorted with both PCT-specific and standard UNIX filters to produce a stream where text lines referring to “similar” code locations are adjacent. A filter can then aggregate counts over text lines with these “similar” suffixes, and hence the similarity determines the level of aggregation. These aggregates are effectively histograms of program counter sam-

ples over the address space of the programs. The histogram bins are determined by the granularity of labels. E.g., function-granularity source coordinates will result in a report of the time spent in various functions.

Users often find it easier to think about time fractions rather than raw sample counts. PCT supports this with a filter that totals the whole text stream and then re-emits it with counts converted to percentages. These percentages are normalized to whatever particular selection of counts is under consideration – either over multiple runs or over multiple objects or other combinations.

The interface for users to these capabilities are simplified by simple shell script wrappers. For example, `pct sym% /tmp/pct/ct/myprogram.*/libc*` will create a function-level profile of time spent in the math library.

There are a few PCT report styles that provide more context around the sampled program locations. PCT can create entire copies of source-code files annotated with either counts or time fractions. We also have an Emacs mode much like `grep-mode` or `compile-mode` to drive examination of `filename:line number` profile reports. This mode allows a user to select report lines of high time fraction and automatically loads a buffer and warps the cursor to that spot in the code.

## 6 Evaluation

A few concerns arise in evaluating any profiling system. First, one must ask if profiling overhead is obtrusive relative to real-time events. Large overhead could make results inaccurate. Bearing in mind that programs being profiled may be quite slow, excessive real-time overhead could dissuade programmers from using the system. A second concern is the accuracy of profiling numbers produced by the system. The following subsections discuss these issues.

### 6.1 Overhead

The overhead of any sampling system is tunably small (or large). There is a fundamental overhead-accuracy trade-off. The more frequently samples are taken, the more overhead incurred by interrupting

the program and recording the samples. However, the larger the sample rate the more accurate a picture one acquires in a given amount of time.

Large sample rates may be desirable. Some programs run only briefly, but are CPU intensive while they run. Accurate probabilities may also motivate a fast sample rate.

On modern CPUs, the cost of signal delivery and resumption of execution system call is typically less than 20  $\mu\text{sec}$ . Thus library-based sampling procedures are very low overhead.

We have measured a gdb-based sampling as taking 500..1000  $\mu\text{sec}$  on 700..1300 MHz Pentium III and Athlon-based systems running Linux, FreeBSD, and OpenBSD. The precise time varies depending on the complexity of parameter lists being decoded, the depth of the stack, the efficiency of the OS, and the CPU. However almost all of this overhead is gdb making many calls to `ptrace()` to reconstruct the argument lists of functions in the backtrace. It is possible to arrange a more minimally informative gdb sampling which does no address translation or decoding. This resulted in under 100  $\mu\text{sec}$ , including the round-trip context switch.

These numbers are still relatively encouraging. For 10 ms sampling granularities the overhead is almost unnoticeably small unless quite rich samples are being taken. At 1 ms sampling rates the overhead starts to become near a factor of two, but overhead is not prohibitive until near 100  $\mu\text{sec}$  rates. This also suggests that a debugging library could result in substantial overhead reduction by computing only the necessary output. Alternately, debugger features could control the verbosity of output more finely.

Also note that, at least on uniprocessors, it does not matter how many processes are being traced. Only one process has the CPU at a time. So some fractional overhead applies to the real time consumed by the entire system of processes.

## 6.2 Accuracy

Sampling overhead does not directly impact the accuracy of time estimates. Any constant amount of sampling overhead has the effect of simply increasing the sampling period. Hence the *variance* of time spent handling signals and recording samples might

impact the accuracy of program counts. In practice, with code that has known time fractions, sampling time variance seems to have a negligible effect.

Assessing precise time fractions of a program creates a need for large sample sizes. The statistics of location counts are approximately binomial. The program is suspended at some location,  $i$ , with probability  $p_i$ . The mean of a binomial random variable for  $N$  trials each of which has probability  $p_i$  is simply  $Np_i$ , while the standard deviation is  $\sqrt{Np_i(1-p_i)}$ . The frequency,  $p_i = n_i/N$  thus has a fractional error proportional to  $1/\sqrt{N}$ . Suppose, for example, location 1 has count  $n_1$  and location 2 has count  $n_2$ . It is easy to show that a two standard deviation test for the condition  $p_1 > p_2$  is approximately  $n_1 - n_2 > 2\sqrt{n_1 + n_2}$ . E.g., to decide  $p_1 > p_2$  for  $n_2 = 1$  requires  $n_1 \geq 5$ . Fortunately, precise probabilities are usually less important than just identifying the expensive areas of a computation.

Reduced real-time performance is the most significant down-side of overhead. For very high sampling rates and very rich data collection a program can run much slower than in its native mode.

Correlations between the time of sampling and paths in the program are more problematic. Consider the specific example of a function which takes almost exactly as long to execute as the time between timer expirations. Also assume this function is repeatedly invoked and dominates the execution time. It should be clear that the program will always be suspended near the same location. Computation is distributed over all the code implementing this function. [17] discusses this problem in the context of CPU usage statistics. DCPI [8] addresses this problem by randomizing the size of the time interval between samples.

While `profil()` is inflexible in this regard, any other PCT collection method optionally uses one-shot timers and re-installs timers with randomly spaced delays in the alarm handler. Using large multiples of a typical 10 ms time quantum are likely to seriously reduce the achieved sample size. However provided that successive periods are unpredictable, even a random alternation between 10 ms and 20 ms guards against a little accidental synchronization. The fixed underlying timer clock makes truly preventing synchronization effects difficult. One cannot build a random interval from even random multiples of a coarse intervals. Synchronization at the scale

of the underlying time quantum could still cause problems. Truly random sample-to-sample intervals clearly require specialized OS support.

## 7 Related Work

Profiling is as old an art as writing programs. As with debugging, there has been a large amount of tool-building and research devoted to automating tasks that were originally done with hand-coded instrumentation. These prior profiling systems all take a more narrow view of profiling than the PCT philosophy of profiling as a *type of debugging* in which programmers apply the same familiar tools and preparations.

Automatic instrumentation introduced the possibility of collecting richer data than classic `profil()`-style samples, such as dynamic call graphs, basic block activations, and control flow arc transitions. Increasing complexity of software systems has driven a need for multi-process and even whole system profiling systems. Both static and dynamic profile-driven optimization have become a focus for those interested in performance.

Additionally, some have considered limited notions of *higher-level* profiling [22] such as the implementation of abstract data types or other alternative algorithm selection. This approach instrumented programs to record, for example, where data is typically inserted into an ordered list. It used this data to decide between array or linked-list representations. Inserts at the beginning favor linked representations while those at the end favor arrays. PCT might be leveraged to answer similar questions without modifying the program to use an instrumented data structure library.

There have been a large number of compile-time automatic instrumentation systems, all of which are invasive, early-bound, and non-extensible. An early hierarchical profiler was `gprof`. [12] The programs `tcov` [6] and `gcov` [1] are similar to `gprof`, but instrument basic blocks instead of function calls. Many implementations of these types of profiler are not robust to improper program exits and are not tolerant of inadequate data in some objects.

Many systems have also implemented some form of link-time instrumentation or post-link-time binary re-writing. [23, 14, 9, 15, 24, 20] These address re-

building issues somewhat and have some weak extensibility. A significant invasion of foreign code may remain, though. The code must be inserted to count executions, or, in more involved cases, log procedure arguments.

Recently, a number of researchers have begun investigating the limits of *dynamic* instrumentation – the re-writing of running executables. IBM has a system called DProbes [18, 10] which enables generic kernel-based late-bound instrumentation. This is similar to our debugger-controller based approach, but aims to build up a toolset for various machines and architectures rather than relying on the existing debugger infrastructure. Much lower overhead would likely be possible via this approach, but a great deal more work would need to be done to allow the sort of arbitrary expressions collectible with debuggers.

Beyond instrumentation systems, there has been significant prior progress in PC-sampling-style profiling as well. There is the classic `prof` [4], and many latter-day counterparts. The most sophisticated system along these lines is probably the Digital Continuous Profiling Infrastructure (DCPI). [8] DCPI has focuses on understanding how the microarchitectural features of Alpha processors play out in full system applications. This system is unfortunately proprietary and non-portable as many of its most impressive features rely upon CPU and OS support. The focus on low-level CPU behavior instead of high-level semantics of programs makes this system more useful for compiler writers and other assembly-level optimizations. For instance it does not support hierarchical call path samples along the lines of. [13]

SGI's IRIX-specific SpeedShop [2] system is probably the closest system in spirit to PCT, though it stops short of a full debugger-profiler. It does sampled hierarchical profiling, has graceful report degradation, and is late-binding. However, in addition to being specific to IRIX on MIPS, it is restricted to dynamically linked executables, and fails to be exhaustive in terms of kernel-resident and late-loaded code. Being proprietary, it is difficult to evaluate its extensibility.

There have been many kernel-level profiling tools as well. Some system call tracers like `strace -c` support simple system call profiling. [7] Yaghmour's Linux Trace Toolkit [25] is a useful kernel-level event monitoring facility. PCT can leverage easily accessible `/proc/profile` data on Linux. PCT's

ptrace()-based techniques do not easily extend to kernel code since the necessary process control features are not typically available on the kernel itself.

The type of non-interactive debugging PCT does is similar to the Expect system for controlling interaction.[16] Our debugger-controller is similar, but with configuration syntax tailored to profiling and the ability to handle large subtrees of processes simultaneously. dbctl also does not rely on the relatively slow logic and string processing of Tcl.[19] As noted earlier, dbctl also allows non-interactive process control other than state sampling.

## 8 Availability

PCT is freely available under an open source license. More information and current software releases can be obtained at the PCT web page:

<http://pdos.lcs.mit.edu/~cblake/pct>

In the realm of simple profiling, every UNIX after AT&T version 7 has support for profil() functionality, which provides at least some capability. Kernel profile integration is currently only available on Linux. The ldd command on OpenBSD and FreeBSD is informative enough to allow tracking shared library usage and process tree profiling.

Generalized profiling should be available on any system with a good source-level debugger for the programming languages of interest. Currently, gdb works well on the above mentioned systems as well as Solaris, HP-UX 9,10,11, AIX, Irix, SunOS, various other BSD's and probably many more platforms. Ports of our interaction scripts to dbx and other debuggers are under way.

## 9 Conclusion

Each year software systems grow in complexity from multiple code regions per address space to multi-process programs. Correctness becomes harder to achieve, and conventional wisdom is to postpone performance analysis as long as possible. PCT requires no more preparation than for debugging. This allows programmers to interleave the optimization

and debugging of their program however they see fit.

PCT unifies access to a number of existing profiling features that have been available for some time and extends profiling in new directions. The PCT debugger-based profiling architecture substantially extends the sort of data that automatic profiling can collect. One can sample programmer-definable, context specific data. Such samples can often more readily expose higher-level algorithmic issues, such as a mismatch between program structures and user inputs. The overhead of PCT scales reasonably with the complexity of the program data being sampled and with sampling rates.

Finally, PCT is very portable by design, requiring no special CPU or OS features or support. Informative data can be gathered on most code in flexible ways. Reports can be generated flexibly based on various data aggregations while the program is still running. These features usually require no recompiling, re-linking, or even re-starting of users' programs.

## References

- [1] gcov: a test coverage program.  
<http://gcc.gnu.org/onlinedocs/gcc-3.0>.
- [2] Irix 6.5 speedshop user's guide.  
<http://techpubs.sgi.com/>.
- [3] ld(1). Unix man pages.
- [4] prof(1). AT&T Bell Laboratories, Murray Hill, N.J., UNIX Programmer's Manual, January 1979.
- [5] profil(2). Unix man pages.
- [6] tcov(1). AT&T Bell Laboratories, Murray Hill, N.J., UNIX Programmer's Manual, January 1979.
- [7] W. Akkerman. strace home page.  
<http://www.liacs.nl/~wichert/strace/>.
- [8] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, R. Sites, M. Vandrvoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4), Nov. 1997.



- [9] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.
- [10] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
- [11] B. Driehuis. ltrace home page. <http://utopia.knoware.nl/users/driehuis/>.
- [12] S. Graham, P. Kessler, and M. McKusick. gprof: a call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982.
- [13] R. J. Hall and A. J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proceedings of the Summer 1993 USENIX Conference: June 21–25, 1993, Cincinnati, Ohio, USA*, pages 1–13, Berkeley, CA, USA, Summer 1993. USENIX.
- [14] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, California, 1991.
- [15] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, La Jolla, California, 18–21 June 1995.
- [16] D. Libes. expect: Curing those uncontrollable fits of interaction. In *Proceedings of the USENIX Summer 1990 Technical Conference*, pages 183–192, Berkeley, CA, USA, June 1990. Usenix Association.
- [17] S. McCanne and C. Torek. A randomized sampling clock for CPU utilization estimation and code profiling. In *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, pages 387–394, Berkeley, CA, USA, Winter 1993. USENIX Association.
- [18] R. J. Moore. A universal dynamic trace for linux and other operating systems. In *Proceedings of the FREENIX Track (FREENIX-01)*, pages 297–308, Berkeley, CA, June 2001. The USENIX Association.
- [19] J. K. Ousterhout. Tcl: An embedable command language. In *Proceedings of the USENIX Association Winter Conference*, 1990.
- [20] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and J.B. Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *The USENIX Windows NT Workshop 1997, August 11–13, 1997, Seattle, Washington*, pages 1–7, Berkeley, CA, USA, Aug. 1997. USENIX.
- [21] A. D. Samples. Profile-driven compilation. Technical Report UCB//CSD-91-627, UC Berkeley, Department of Computer Science, 1991.
- [22] A. D. Samples. Compiler implementation of ADTs using profile data. 641, 1992.
- [23] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, Computer System Lab, Nov. 1991.
- [24] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *ACM SIGPLAN Notices*, 29(6):196–205, June 1994. ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI).
- [25] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 13–26, Berkeley, CA, June 18–23 2000. USENIX Association.



# Engineering a Differencing and Compression Data Format

David G. Korn and Kiem-Phong Vo  
AT&T Laboratories – Research  
180 Park Avenue, Florham Park, NJ 07932, U.S.A.  
dgk, kpv@research.att.com

## Abstract

Compression and differencing techniques can greatly improve storage and transmission of files and file versions. Since files are often transported across machines with distinct architectures and performance characteristics, compressed data should be encoded in a form that is portable and efficient to decode. This paper describes the Vcdiff encoding format for differencing and compression data and presents an empirical study showing its effectiveness.

## 1 Introduction

*Data differencing* computes a compact transformation to take a source file to a target file based on their differences. *Data compression* compresses data in a single file. The UNIX utility *diff* is an example data differencing tool while *compress* and *gzip* are well-known data compressors. Differencing and compressed data are good for storage and transmission as they are often much smaller than the originals. Differencing and compression techniques are traditionally treated as distinct forms of data processing. Our work on the Vdelta compressor [3, 4] showed that compression and differencing can be treated uniformly by unifying the Lempel-Ziv'77 string parsing scheme [14] and Tichy's block-move technique [12]. This unification is called *delta compression*.

Compressed data need to be encoded in a portable and efficient format so that they can be transported across a network such as the Internet which consists of diverse hardware and software platforms. Many compressors are available, each with its own way to represent data. However, little has been published on the encoding formats used by these compressors. A notable exception is *gzip* whose encoding format is published in the IETF Standard *Deflate* [1]. Data differencing is much less developed than data compression so there are few tools available. The *diff* utility only works on text files and outputs editing commands to be processed by the UNIX line editor *ed*. The only published format for differencing of binary data is the W3C Standard *Gdiff* [13]. Outside of this work, there is no published encoding format for delta compression, i.e., a format suitable for both compression and differencing. Given the intended applications, we stipulate that such a data encoding format should have the following attributes:

- *Algorithm independence*: The encoding format must be independent from the algorithms used to compress data. This allows a receiver to decode compressed data without having to know how it was computed.
- *Data portability*: The encoding format must be free from hardware architecture issues such as byte order and word size. This allows a receiver to decode data without knowing the architecture of the encoding machine.
- *Output compactness*: The encoding format must compactly represent compressed and delta data. It should also be transparently extensible by encoders to maximize compression efficiency.
- *Decoding efficiency*: The encoding format must be decodable on machines with limited computational power and memory. This is important for web-based applications with small clients such as PCs or hand-held devices.

The mentioned Vdelta software was instrumental in the work to extend HTTP1.1 for Delta Encoding [9, 10]. However, the encoding format used by Vdelta was not sufficiently compact for compression data (i.e., when single files are compressed) and not easily extensible. Since then, we have designed a new format Vcdiff for delta compression. This format incorporates a number of innovations that enable compact data representation with extensibility to exploit application-specific knowledge in gaining further compression. This paper discusses the essential elements of the Vcdiff encoding format and presents performance data showing its effectiveness. A full description of the format is given in a current IETF Proposed Standard [6].

## 2 Algorithm independence

Techniques such as Vdelta, Lempel-Ziv and block-move are based on *string matching algorithms* [4, 7, 8] to find matches either across files or in the same file. Each such match can then be compactly encoded by its location and length. Different string matching algorithms will typically find different matches.

String matching algorithms often stress memory resources so, on current computers, they are not effective for processing large files in the order of hundreds of megabytes or gigabytes. To deal with this, a target file can be partitioned into sufficiently small contiguous segments of data called *target windows*, each of which is to be compressed separately. To improve compression, such a target window may be compared against some *source window*, a contiguous segment of data from either the source file or the target file itself. In the latter case, the source window is required to come from some part of the target file preceding the current target window so that, during decoding, the data for such a window is well-defined. Finding the right matching source window for a given target window is crucial for compressing data. Algorithms to do this are called *windowing algorithms*.

String matching and windowing algorithms clearly affect compression effectiveness. However, from the point of view of designing an encoding format, it is preferable to abstract away the details of such algorithms. In this way, simple and generic decoders can be constructed without knowing how the data was encoded. An additional benefit is that software vendors and/or researchers can continue improving the encoding algorithms without affecting the receivers of compressed data. We discuss how to design such an encoding format next.

### 2.1 Windowing data

Source and target windows may have different sizes but their sizes are chosen so that they can be processed entirely in memory. For data differencing, the traditional method simply aligns source and target windows by file offsets. For data compression, the popular rolling window method uses a small data segment immediately before the target window as the source window. These algorithms work well with small files since the window choices are limited (so they are mostly right by fiat) but they are suboptimal for large files as matching data may occur randomly and much further apart. In a work-in-progress, Vo explored a content-based method [11] to find source windows that would likely match well with given target windows (Section 5). Regardless of what window selection algorithm is used, a decoder does not need to know about it as long as the encoding format records the following data about source windows:

- An indication of whether the source window is from the source file or the target file,
- The starting position of the source window in the respective file, and
- The length of the source window.

Given this basic data, a decoder can obtain the appropriate source data to be used with the compressed data to decode a target window. Next we discuss what comprises compressed data.

### 2.2 String matching and delta instructions

When a target window  $T$  with size  $t$  is compressed given a source window  $S$  of size  $s$ , we shall think of  $S$  and  $T$  as substrings of a superstring  $U$  formed by concatenating them like this:

$$S_0S_1\dots S_{s-1}T_0T_1\dots T_{t-1}$$

The *address* of a byte in  $S$  or  $T$  is referred to by its location in  $U$ . Thus, for any  $k \leq t$ , the address of  $T_k$  is  $s + k$ . The compressed data consists of a sequence of instructions called *delta instructions*. There are three types:

- **ADD:** This instruction has two arguments, a size  $\lambda$  and a sequence of  $\lambda$  bytes to be copied.
- **COPY:** This instruction has two arguments, a size  $\lambda$  and an address  $\alpha$  in the string  $U$ . These arguments specify the substring of  $U$  that must be copied into the target window being constructed. For programming convenience, we assert that such a substring must be entirely contained in either  $S$  or  $T$ .
- **RUN:** This instruction has two arguments, a size  $\lambda$  and a byte that will be copied  $\lambda$  times.

Let  $\lambda(i)$  be the size of any delta instruction  $i$  and  $\alpha(i)$  the associated address if  $i$  is a COPY. Let  $I = i_1i_2\dots i_n$  be a sequence of delta instructions. Then each instruction  $i_k$  encodes a data segment  $\sigma(i_k)$  of size  $\lambda(i_k)$ . Let  $p = \sum_{1 \leq m \leq k-1} \lambda(i_m)$ . We say that  $I$  is a *faithful* representation of  $T$  if:

- For all  $k$ ,  $\sigma(i_k)$  is equal to the substring of  $T$  starting at  $p$  with size  $\lambda(i_k)$ ;
- If  $i_k$  is a COPY, then  $\alpha(i_k) < p + s$ ; and
- $\sum_{1 \leq m \leq n} \lambda(i_m)$  is equal to the size of  $T$ .

1. Set  $p = 0$  and  $k = 0$ .
2. If  $i_k$  is a RUN instruction, copy the associated data byte  $\lambda(i_k)$  times to  $T$  starting at  $p$ .
3. If  $i_k$  is an ADD instruction, copy the associated data to  $T$  starting at  $p$ .
4. If  $i_k$  is a COPY instruction,
  - (a) If  $\alpha(i_k) < s$ , copy  $\lambda(i_k)$  bytes from  $S$  starting at  $\alpha(i_k)$  to  $T$  starting at  $p$ ;
  - (b) Else, copy  $\lambda(i_k)$  bytes from  $T$  starting at  $\alpha(i_k) - s$  to  $T$  starting at  $p$ .
5. Set  $p = p + \lambda(i_k)$  and  $k = k + 1$ .
6. If  $k \leq n$ , go to 2.

Figure 1: Decoding delta instructions

Let  $S$  be a source string of size  $s$ ,  $T$  a target window of size  $t$  and  $I$  a faithful representation of  $T$ . Figure 1 shows the algorithm to reconstruct  $T$  from  $I$  and  $S$ . A string copy operation is assumed to be carried out from left to right so that Step 4.b is well-defined. Since the total running time of the algorithm is proportional to the number of bytes copied, the below result immediately follows from the definition of a faithful representation:

**Theorem 1** *A target window encoded with a faithful sequence of delta instructions can be decoded in  $O(t)$  time and space where  $t$  is the size of the window.*

```

S: abcdefghijklmnop
T: abcdwxyzefghefghfghfghzzzz

COPY  4, 0
ADD    4, wxyz
COPY   4, 4
COPY  12, 24
RUN    4, z

```

Figure 2: Delta instructions transforming  $S$  into  $T$

Figure 2 shows example source and target windows and a sequence of delta instructions encoding the target data. It is easy to verify that this sequence is a faithful representation of  $T$ . The first COPY instruction copies 4 bytes from address 0, i.e., the string *abcd* in the source window  $S$ . Next is an ADD instruction that adds the 4

specified bytes *wxyz*. Note that the fourth instruction copies data from  $T$  itself since address 24 is position 8 in  $T$ . This instruction also shows that the data to be copied can overlap with the data being copied from as long as the latter starts earlier. This enables efficient encoding of periodic sequences, i.e., sequences with regularly repeated subsequences. The final RUN instruction compactly encodes the last four bytes of  $T$ .

Given a pair of target and source windows, there are usually many different faithful representations of the target data. For example, the target data in Figure 2 can also be faithfully represented with a single ADD instruction that includes all the data. From a compression point of view, it is desirable to find the representation that requires the least number of bytes to encode. Unfortunately, this problem is NP-hard even when sizes and addresses are encoded with some fixed number of bits (SR22 and SR23 in Garey and Johnson [2]). This situation improves when relaxed to just finding the minimum number of delta instructions without worrying about whether or not their encoding minimizes the compressed output. In this case, greedy approaches such as Lempel-Ziv parsing or Tichy block-move [12] do compute the minimal number of delta instructions in linear time and space given appropriate string matching algorithms [7, 8]. The Vdelta algorithm [4] relaxes this minimality to trade for faster string matching and less working memory. In any case, the point with delta instructions is that, no matter how they are computed, Theorem 1 guarantees that a generic decoder can be written that always runs in linear time and space.

### 3 Data portability

The Vcdiff encoding format is byte-oriented. Each byte is limited to its lower eight bits for portability. The bits in a byte are ordered from right to left so that the least significant bit (LSB) has value 1, and the most significant bit (MSB), has value 128.

Sizes and file offsets are unsigned integers encoded via a portable variable-sized format (originally introduced in the Sfiio library [5]). This encoding treats an unsigned integer as a number in base 128. Then, each digit in this representation is encoded in the lower seven bits of a byte. Except for the least significant byte, other bytes have their most significant bit turned on to indicate that there are still more digits in the encoding. The two key properties of this integer encoding that are beneficial to a data compression format are:

- The encoding is portable among systems using 8-bit bytes, and
- Small values are encoded compactly.



Below is the encoding of the integer 123456789 in four 7-bit digits whose values are 58, 111, 26, 21 in order from most to least significant. In the 8-bit representation of these digits, the MSBs of 58, 111 and 26 are on.

10111010	11101111	10011010	00010101
MSB+58	MSB+111	MSB+26	0+21

## 4 Encoding delta instructions

The delta instructions represent string matching results. In data differencing applications of text files, changes between source and target data are often small, resulting in long common substrings. When that is the case, any straightforward representation of the delta instructions would be adequate. However, for differencing of binary files or general compression, matched substrings are often short so that the delta instructions must be encoded well to achieve good compression rates. The key to compact encoding revolves around the questions of how to encode addresses of COPY instructions efficiently and how to deal with instructions having small sizes or limited number of sizes. This leads to the ideas of *address encoding modes* and *instruction code tables* which are discussed next.

### 4.1 Address caches and encoding modes

Data in local regions are often replicated with minor changes. This is especially true in data differencing where target files are created from small changes in source files. Thus, the addresses of successive COPY instructions often occur close by or even exactly equal to one another. To take advantage of this phenomenon, Vcdiff maintains two types of address caches:

- A *near* cache is an array with *s\_near* slots of previously matched addresses. An address *p* can be encoded against a cached address *q* as  $p - q$  if  $p \geq q$ .
- A *same* cache is an array with  $s\_same * 256$  slots of previously matched addresses. If an address *p* is equal to  $same[p\%(s\_same * 256)]$ , then *p* can be encoded with the single byte value  $p\%256$ .

It is clear that an encoder and a decoder must be in synch with respect to maintaining the address caches. The protocol to enforce this is as follows:

1. Before processing (i.e., encoding or decoding) a target window, all cache slots are initialized to zero.
2. After processing each COPY instruction, its address *p* is used to update the caches as follows:

- (a) The slots in the *near* cache are managed as a circular buffer with a current *index*. The address *p* is first added to  $near[index]$ , then *index* is incremented modulo *s\_near*.
- (b) The *same* cache is a hash table of size  $s\_same * 256$ . The address *p* is added to  $same[p\%(s\_same * 256)]$ .

In the above cache usage, the *address encoding mode*, i.e., the manner in which the address *p* of a COPY instruction is encoded must be recorded in the encoding data. Let *here* be the current position in the target data (i.e., the start of the data about to be encoded or decoded). Below are the address modes:

- VCD.SELF: This mode has value 0 and indicates that *p* was encoded as itself.
- VCD.HERE: This mode has value 1 and indicates that *p* was encoded as *here - p*.
- *Near*: There are *s\_near* modes in the range  $[2, s\_near + 1]$ . If *m* is the mode of the address encoding then *p* was encoded as  $p - near[m - 2]$ .
- *Same*: There are *s\_same* modes in the range  $[s\_near + 2, s\_near + s\_same + 1]$ . If *m* is the mode of the encoding then *p* was encoded as a single byte *b* such that  $same[(m - (s\_near + 2)) * 256 + b]$  is equal to *p*.

By default, Vcdiffuses 4 for *s\_near* and 3 for *s\_same* resulting in a total of 9 different addressing modes.

### 4.2 Instruction code tables

Successive delta instructions often represent short matches separated by small amounts of unmatched data. So the sizes of the COPY and ADD instructions are often small. This is particularly true of binary data such as executable files or semi-structured data such as HTML or XML. In such cases, it is beneficial to combine sizes, instruction types and even successive pairs of instructions. The effectiveness of such combinations depend on many factors including the data being processed and the string matching algorithm in use. For example, in a case where many COPY instructions with the same data sizes are generated, it may be worth encoding these instructions more compactly than others.

To maintain independence from the choices made in encoding algorithms, we introduce the notion of *instruction code tables*, each of which consists of 256 entries. These entries describe combinations of sizes, instruction types and pairs of instructions. The encoder and decoder(s) of a compressed dataset must share the same

table. Then, the encoding only records the indices of the table entries, each of which fits in a single byte.

As depicted below, an entry in an instruction code table conceptually consists of two triples, each of the form (inst, size, mode):

inst1	size1	mode1	inst2	size2	mode2
-------	-------	-------	-------	-------	-------

- *inst*: This field can be one of: NOOP, ADD, RUN or COPY to indicate the instruction types. NOOP means that no instruction is specified.
- *size*: This field is either zero or positive. Zero means that the size associated with the instruction is encoded separately in the encoding data. A positive value defines the actual data size so the encoding data will omit it.
- *mode*: This field is significant only when the instruction type is COPY. It defines the encoding mode used to encode the associated address.

Thus, each entry in the instruction code table can encode a single instruction (one of the triples is a NOOP) or two successive instructions. Vcdiff itself defines a *default instruction code table* for the case when the *near* cache has 4 slots and the *same* cache has  $3 * 256$  slots. Thus, there are 9 address modes for COPY instructions. The first two are VCD\_SELF(0) and VCD\_HERE(1). Modes 2, 3, 4 and 5 are for addresses coded against the *near* cache. And, modes 6, 7 and 8 are for addresses coded against the *same* cache. This default table is assumed to be available with each encoder and decoder. The Vcdiff encoding format also allows an encoder to define its own custom code table but then it has to encode this table in the data itself [6].

Table 1 depicts the default instruction code table. Each numbered line represents one or more entries (recall that an entry in the instruction code table may represent up to two combined delta instructions). The last column ("Index") shows which index value or range of index values of the entries covered by that line. The first 6 columns of a line in the depiction describe the pairs of instructions used for the corresponding index value(s). For example, line 1 shows the single RUN instruction with index 0. As the size field is 0, this RUN instruction always has its actual size encoded separately in the encoding data. Line 2 shows the 18 single ADD instructions. The ADD instruction with size field 0 (i.e., the actual size is coded separately) has index 1. ADD instructions with sizes from 1 to 17 use code indices 2 to 18 and their sizes are as given (so they will not be separately encoded). Lines 12 to 21 show the pairs of instructions that are combined together. For example, line 12 depicts the 12 entries in which an ADD instruction is combined with

an immediately following COPY instruction. The entries with indices 163, 164, 165 represent the pairs in which the ADD instructions all have size 1 while the COPY instructions have mode VCD\_SELF(0) and sizes 4, 5 and 6 respectively.

Table 2 shows two different encodings of the delta instructions from Figure 2: Plain and Optimized. In the Plain encoding, each instruction was simply encoded. For example, the first COPY instruction used code index 19 so its size and address were separately encoded entailing a total cost of three bytes. Similarly, the ADD instruction used index 1 with separately encoded size so the cost was 6 bytes. Altogether, the Plain coding used 18 bytes which substantially improved over the original data size of 28 but was not optimal.

In the Optimized encoding, the first COPY instruction used code index 20 with implicit size 4. Thus, its encoding took only two bytes. The second and third instructions, ADD and COPY, were combined via code index 172 with both sizes implicitly defined. Thus, both instructions were encoded in 6 bytes instead of the original 9 bytes in the Plain encoding. Altogether, the size of the Optimized encoding improved to 13 bytes.

The Optimized encoding shows that judicious use of instructions with implicitly defined sizes and combined instructions can substantially improve the compression rate. We discuss next how to compute the optimal encoding given a fixed code table.

### 4.3 Optimizing instruction encoding

Section 4.2 showed that an encoder has a wide latitude in choosing when and how to combine and encode delta instructions. In fact, for any fixed instruction code table, one can optimize the encoding of a sequence of delta instructions using dynamic programming. Toward this end, let  $I = i_1 i_2 \dots i_n$  be a sequence of delta instructions. We shall use  $I_k$  to denote the subsequence of  $I$  starting from the  $k^{th}$  instruction and extending to the end of  $I$ . For example,  $I = I_1$ . We define  $I_k$  to be the empty sequence whenever  $k > n$ .

The code entries in an instruction code table assumes that the addresses of COPY instructions and the data of ADD and RUN instructions are always coded separately. Thus, to optimize the encoding, we only need to consider the sizes of the instructions and their types, i.e., ADD, RUN, COPY and any addressing modes. Now, for each instruction  $i$ , let the cost of  $i$ ,  $c(i)$ , be the number of bytes required to encode  $i$  and its size using the best choice from the instruction code table. We assume that the instruction code table has been defined so that there is at least one way for doing this. Likewise, for any two consecutive instructions  $i$  and  $j$ , let the cost  $c(i, j)$  be the number of bytes required using the best table entry that

Table 1: The default instruction code table

	Type	Size	Mode	Type	Size	Mode	Index
1	RUN	0	0	NOOP	0	0	0
2	ADD	0, [1,17]	0	NOOP	0	0	[1,18]
3	COPY	0, [4,18]	0	NOOP	0	0	[19,34]
4	COPY	0, [4,18]	1	NOOP	0	0	[35,50]
5	COPY	0, [4,18]	2	NOOP	0	0	[51,66]
6	COPY	0, [4,18]	3	NOOP	0	0	[67,82]
7	COPY	0, [4,18]	4	NOOP	0	0	[83,98]
8	COPY	0, [4,18]	5	NOOP	0	0	[99,114]
9	COPY	0, [4,18]	6	NOOP	0	0	[115,130]
10	COPY	0, [4,18]	7	NOOP	0	0	[131,146]
11	COPY	0, [4,18]	8	NOOP	0	0	[147,162]
12	ADD	[1,4]	0	COPY	[4,6]	0	[163,174]
13	ADD	[1,4]	0	COPY	[4,6]	1	[175,186]
14	ADD	[1,4]	0	COPY	[4,6]	2	[187,198]
15	ADD	[1,4]	0	COPY	[4,6]	3	[199,210]
16	ADD	[1,4]	0	COPY	[4,6]	4	[211,222]
17	ADD	[1,4]	0	COPY	[4,6]	5	[223,234]
18	ADD	[1,4]	0	COPY	4	6	[235,238]
19	ADD	[1,4]	0	COPY	4	7	[239,242]
20	ADD	[1,4]	0	COPY	4	8	[243,246]
21	COPY	4	[0,8]	ADD	1	0	[247,255]

Table 2: Encoding the delta instructions in Figure 2

Delta Inst.	Plain	Optimized
COPY 4, 0	19 4 0	20 0
ADD 4, wxyz	1 4 wxyz	172 wxyz 4
COPY 4, 4	19 4 4	
COPY 12, 24	19 12 24	28 24
RUN 4, z	0 4 z	0 4 z

combines both instructions. If there is no way to combine  $i$  and  $j$ , we let  $c(i, j)$  be infinite. Finally, let  $C(I)$  be the cost of encoding the sequence  $I$ . Then,  $C(I)$  can be obtained by solving the following dynamic program:

$$C(I) = \begin{cases} 0 & \text{if } I = \phi; \\ c(i_1) & \text{if } |I| = 1; \text{ otherwise,} \\ \min\{c(i_1) + C(I_2), c(i_1, i_2) + C(I_3)\}. \end{cases}$$

The first case states that the cost of encoding an empty sequence is 0. The second case states the cost of encoding a sequence with a single instruction. The last case computes the optimal cost by minimizing between two alternatives: encoding the first instruction by itself or combining the first two instructions. In each alternative,

recursion is used to deal with the rest of the sequence.

Since each  $C(I_k)$  is uniquely determined by the sequence  $I_k$ , we can keep track of all processed subsequences in  $O(|I|)$  space so that the recursion can be pruned whenever it arrives at a processed subsequence. We have shown:

**Theorem 2** *Given a fixed instruction code table, any sequence of delta instructions  $I$  can be encoded optimally in  $O(|I|)$  time and space.*

In addition to optimizing delta instruction encoding given a fixed code table, it is also possible for an application to define its own code tables inside the encoding data [6]. This enables an application to gain further compression by specially treating certain instructions or

pairs of instructions that are much more popular than others. We are investigating the question of how to compute such an optimal instruction code table. The mentioned IETF Proposed Standard [6] also discusses the use of secondary compressors to further compress the instruction encoding.

## 5 Performance

We show the effectiveness of the Vcdiff encoding format in two ways. The first set of experiments was based on three different source code archives of the Gnu C compiler, gcc-2.95.1.tar, gcc-2.95.2.tar and gcc-2.95.3.tar. We used the Vcodex/Vcdiff software (Section 8) to compare various options of Vcdiff against the *gzip* and *compress* tools. These files were very large so that some windowing scheme must be used. In the second set of experiments, we collected the home page of www.cnn.com every hour for 10 days and computed the deltas using various methods.

### 5.1 Comparing with *compress* and *gzip*

We compared Vcdiff against *compress* and *gzip* using the mentioned three source code archives of the GNU C compiler. The experiments were done on an SGI-MIPS3, 400MHZ. Timing results were obtained by running each program three times and taking the average of the total cpu+system times. Below are the different Vcdiff runs:

- *Vcdiff-c*: Vcdiff was used for compression only. That is, no source file was used. This directly compared Vcdiff, *gzip* and *compress* as compressors.
- *Vcdiff-d*: Vcdiff was used for differencing only. That is, matching was allowed only between source and target data. Windows were simply matched by positions across target and source files.
- *Vcdiff-dc*: This is similar to Vcdiff-d but matching within target data, was allowed, i.e., delta compression was used.
- *Vcdiff-dcw*: This is similar to Vcdiff-dc but a content-based windowing algorithm [11] was used to select source windows more likely to match with given target windows. Thus, file offsets of source and target windows would seldom align.

Table 3 shows the experimental results. Note that compression times were typically dominated by the string matching and encoding algorithms. For example, the large time variation in the Vcdiff rows was strictly due to the windowing and string matching algorithms used in the Vcodex/Vcdiff software. Such measurements were

somewhat irrelevant from the point of view of evaluating an algorithm-independent encoding format. However, the decompression times were indicative of how the different formats would perform in practice.

The pure compressor Vcdiff-c gave worse compression rate than *gzip* but better than *compress*. However, it always decompressed fastest. Version gcc.2.95.2 was similar to version gcc.2.95.1. Thus, compressing gcc.2.95.2 given gcc.2.95.1 gained up to a factor of 500 in size reduction as shown in the last three rows. On the other hand, the files in the archive gcc.2.95.3 were sufficiently changed and rearranged from gcc.2.95.2 so that simply matching source and target windows by file positions were ineffective. As a result, Vcdiff-d and Vcdiff-dc did not perform well even though delta compression did help Vcdiff-dc to beat Vcdiff-c and come close to *gzip*. Vcdiff-dcw still worked well due to the content-based windowing algorithm. There was a clear time cost for using such an algorithm during encoding but decoding time was not affected.

Finally, the *cat* row of the table shows the times required to just copy the files gcc.2.95.2.tar and gcc.2.95.3.tar, respectively 1.08 and 1.05 seconds. Thus, in the best case of Vcdiff-dcw, decompression times were only about 70-90% worst than plain copying of the data. The dramatic size reduction meant that, with an appropriate encoder, the Vcdiff encoding format presented a good mechanism for transporting data without taxing client machines on decoding.

### 5.2 Compressing a set of web pages

We collected the home pages of www.cnn.com every hour starting at 12:00AM on 10/23/2001 and ending at 11:00PM 11/01/2001. The below methods for delta compression were used:

- *diff+gzip*: This method runs the *diff -e* program to compute the differences, then pipes the result to *gzip* for further compression.
- *gdiff*: We instrumented the Vcodex/Vcdiff software to run the Vcdiff string matching algorithm but output results in the *Gdiff* format.
- *gdiff+gzip*: This is like the above but the result is piped to *gzip* for further compression.
- *vcdiff*: This uses the Vcdiff encoding format.

We ran two different experiments. In the *First* experiment, each file is compressed against the first file collected while, in the *Successive* experiment, each file is compressed against the one in the previous hour. Table 4 summarizes the compression results. The *raw* row shows

Table 3: Comparing *Vcdiff* to *gzip* and *compress* using the gcc-2.95.[123] archives

Compressor	2.95.2 (55,797,760)			2.95.3 (55,787,520)		
	Size	Comp.(s)	Decomp.(s)	Size	Comp.(s)	Decomp.(s)
cat	55,797,760	1.08	1.08	55,787,520	1.05	1.05
compress	19,939,390	13.85	7.09	19,939,453	13.54	7.00
gzip	12,973,443	42.99	5.35	12,998,097	42.63	5.62
Vcdiff-c	15,358,786	20.04	4.65	15,371,737	20.09	4.74
Compressor	2.95.2 given 2.95.1 (55,746,560)			2.95.3 given 2.95.2 (55,797,760)		
	Size	Comp.(s)	Decomp.(s)	Size	Comp.(s)	Decomp.(s)
Vcdiff-d	100,971	10.93	1.92	26,383,849	71.41	6.41
Vcdiff-dc	97,246	20.03	1.84	14,461,203	42.48	4.82
Vcdiff-dcw	256,445	44.81	1.84	1,248,543	61.18	1.99

Table 4: Delta compression of www.cnn.com

Method	First			Successive		
	Min.	Max.	Avg.	Min.	Max.	Avg.
raw	44,602	50,033	46,036	44,602	50,033	46,036
diff+gzip	20	6,257	4,955	20	3,146	1,017
gdiff	11	5,597	4,277	11	1,767	458
gdiff+gzip	31	4,335	3,336	31	1,496	434
vcdiff	23	4,249	3,274	23	1,423	385

the Minimum, Maximum and Average sizes of the collected files. Later rows show the same statistics for the compressed data. Delta compression was effective in reducing the data sizes overall. The best method, *vcdiff* reduced data by a factor of about 15 in the *First* experiment and about 120 in the *Successive* experiment.

Figure 3 shows in detail the sizes of the compressed data in the *First* experiment. The order of the methods from worst to best was *diff+gzip*, *gdiff*, *gdiff+gzip* and *vcdiff*. Since *gdiff* and *vcdiff* were based on the same underlying algorithms to compute delta instructions, the results compared directly the effectiveness of the different encoding formats. Even with the additional compression step using *gzip* (i.e., the *Deflate* format) the *gdiff+gzip* results were still slightly worse than *vcdiff*. The fact that delta compression was still effective after a fairly long duration of 10 days suggested that these pages were generated from some large template that seldom changed.

Figure 4 shows results from the *Successive* experiment. The *diff+gzip* data fluctuated wildly because *diff* was line-oriented and could not handle small changes made on many lines. Due to this large fluctuation, the format used in Figure 3 did not show the data well. Therefore, we plotted all data points relative to the ones from *vcdiff* by simply dividing each data point by the corresponding value from *vcdiff*. Thus, the flat horizontal

line at 1 represented *vcdiff* data. The encoding formats of *gzip*, *gdiff* and *vcdiff* required fixed overhead sets of bytes (shown in the *Min.* columns in Table 4 as the compared files were identical in that case). We subtracted such overheads from the data points before dividing to remove the large distortion in the ratios when files changed little. *Vcdiff* was again the best encoding format for delta compression in this experiment. In fact, Table 4 showed that it typically reduced data by more than 2 orders of magnitude since files changed very little in successive hours.

Timing results were not shown in the above experiments but *diff+gzip* and *gdiff+gzip* were much slower than *vcdiff* and *gdiff* because of the use of multiple processes and, in the case of *diff+gzip*, slow text alignment algorithms. For *vcdiff* and *gdiff*, it was also hard to obtain meaningful measurements since the files were small and the encoding algorithms were sufficiently fast so that process start-up time became the dominating factor.

## 6 Summary

We described *Vcdiff*, a general and portable encoding format for delta compression, i.e., combined compression and differencing. This is the first fully described encoding format for this type of data processing. *Vcdiff* introduced the novel idea of an instruction code ta-



ble to allow combining delta instructions to optimize compression rate. We showed how to compute minimal encodings given a fixed code table using dynamic programming. More importantly, the nature of the encoding format enables construction of decoders free from any knowledge of encoders and guaranteed to run in linear time and space. Thus, Vcdiff is suitable for web-based client-server applications in which a big server can send data to much smaller clients with different hardware architectures. We presented performance results showing that Vcdiff compares favorably to other formats for data differencing including the W3C Gdiff Standard and the use of *diff* and *gzip*. Vcdiff is the subject of a current IETF Proposed Standard [6].

## 7 Acknowledgements

Thanks are due to Balachander Krishnamurthy, Jeff Mogul and Arthur Van Hoff who provided much encouragement to publicize Vcdiff.

## 8 Code availability

The Vcdiff data format described here is free from any patent claims. An implementation of Vcdiff is available as a part of the Vcodex package written by Phong Vo. The code can be obtained from <http://www.research.att.com/sw/tools>.

## References

- [1] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. In <http://www.ietf.org>. IETF RFC1951, 1996.
- [2] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, NY, 1979.
- [3] J.J. Hunt, K.-P. Vo, and W.F. Tichy. An Empirical Study of Delta Algorithms. In *IEEE Software Configuration and Maintenance Workshop*, 1996.
- [4] J.J. Hunt, K.-P. Vo, and W.F. Tichy. Delta Algorithms: An Empirical Analysis. *ACM Transactions on Software Engineering and Methodology*, 7:192–214, 1998.
- [5] D.G. Korn and K.-P. Vo. SFIO: Safe/Fast String/File IO. In *Proc. of the Summer '91 Usenix Conference*, pages 235–256. USENIX, 1991.
- [6] D.G. Korn and K.-P. Vo. The VCDIFF Generic Differencing and Compression Data Format. [www.research.att.com/sw/tools/vcodex](http://www.research.att.com/sw/tools/vcodex), [www.ietf.org](http://www.ietf.org), 2000.
- [7] U. Manber and G. Meyer. Suffix Array: A New Method for On-Line String Searches. *SIAM J. Computing*, 22:935–948, 1993.
- [8] E. M. McCreight. A space-economical suffix tree construction algorithm. *JACM*, 23:262–272, 1976.
- [9] J.C. Mogul, F. Douglass, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *SIGCOMM '97, Cannes, France*, 1997.
- [10] J.C. Mogul, B. Krishnamurthy, F. Douglass, A. Feldmann, Y. Goland, and A. Van Hoff. Delta Encoding in HTTP. In *draft-mogul-http-delta-10*. IETF, 2000.
- [11] S. Suri and K.-P. Vo. Effective Windowing for Delta Compression. Work-in-progress, 2001.
- [12] Walter F. Tichy. The String-to-String Correction Problem with Block Moves. *ACM Transactions on Computer Systems*, 2(4):309–321, November 1984.
- [13] A. van Hoff and J. Payne. Generic Diff Format Specification. In <http://www.w3.org/TR/NOTE-gdiff-19970825.html>, 1997.
- [14] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

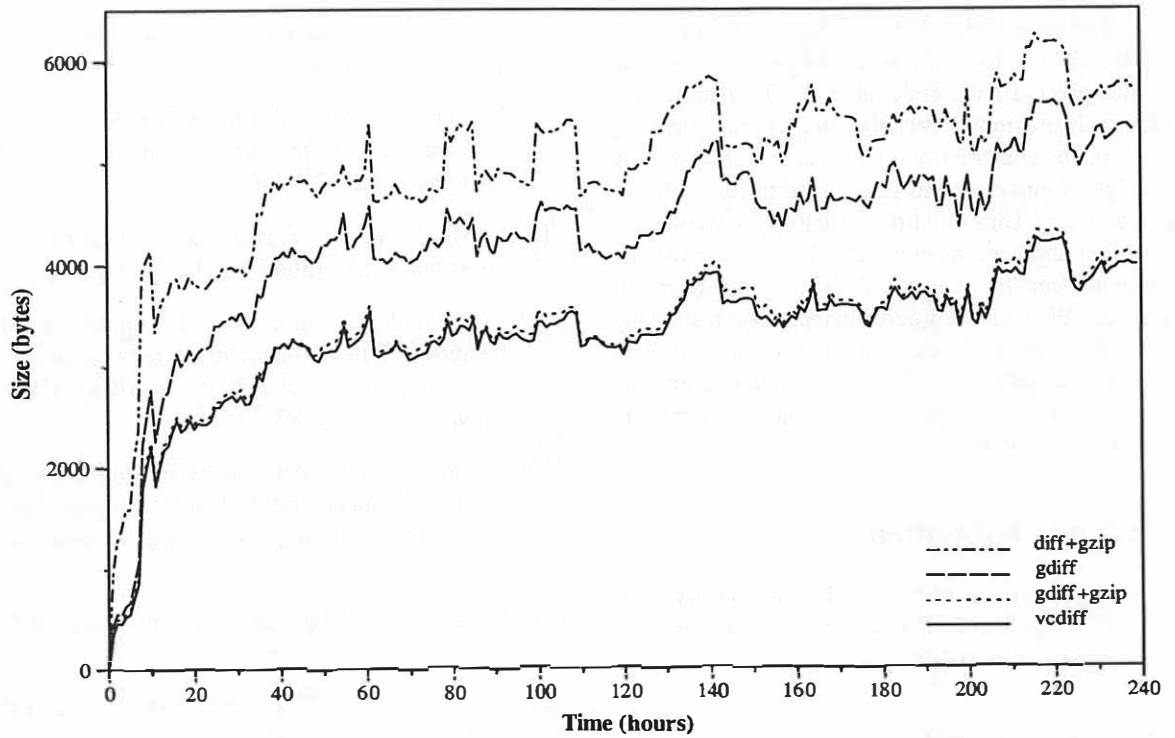


Figure 3: Delta compression of www.cnn.com against a fixed first page

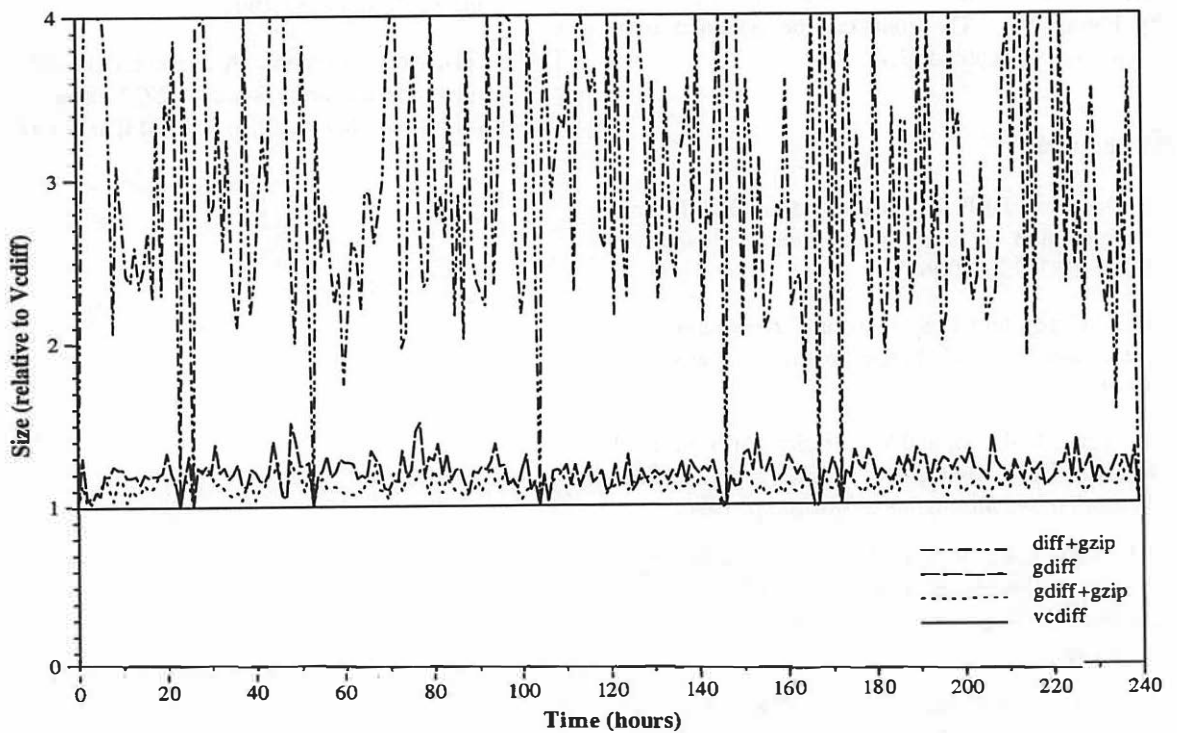


Figure 4: Delta compression of www.cnn.com in successive hours

# A Precise and Efficient Evaluation of the Proximity between Web Clients and their Local DNS Servers

Zhuoqing Morley Mao\*, Charles D. Cranor, Fred Douglass†, Michael Rabinovich,  
Oliver Spatscheck, and Jia Wang  
*AT&T Labs—Research*

## Abstract

Content Distribution Networks (CDNs) attempt to improve Web performance by delivering Web content to end-users from servers located at the edge of the network. An important factor contributing to the performance improvement is the ability of a CDN to select servers in the proximity of the requesting clients. Most CDNs today use the Domain Name System (DNS) to make such server selection decisions. However, DNS provides only the IP address of the client's local DNS server to the CDN, rather than the client's IP address. Therefore, CDNs using DNS-based server selection assume that clients are "close" to their local DNS servers.

To quantify the proximity between clients and their local DNS servers, we propose a novel, precise, and efficient technique for finding the associations of client to local DNS servers. We collected more than 4.2 million such unique associations in three months. From this data, we study the impact of proximity on DNS-based server selection using four different proximity metrics. We conclude that DNS is good for very coarse-grained server selection, since 64% of the associations belong to the same Autonomous System. DNS is less useful for finer-grained server selection, since only 16% of the client and local DNS associations are in the same *network-aware cluster* [13] (based on BGP routing information from a wide set of routers). As an application of this methodology, we evaluate DNS-based server selection in three of the largest commercially deployed CDNs to study its accuracy.

## 1 Introduction

Creating and managing a high-performance, Internet-scale Web service is a formidable challenge involving

deployment of multiple Web servers in strategic locations throughout the network. The introduction of Content Distribution Networks (CDNs) has allowed organizations to overcome this challenge by outsourcing the distribution of their Web content. With CDNs, content providers need only to supply an origin Web server — the CDN distributes the content to end users through a set of CDN servers it has deployed in the network. Ideally, this reduces Web response time and download latencies in addition to providing overload protection and bandwidth savings.

In a well-designed CDN, servers are placed to avoid congested links and slow network paths. When a Web client requests content, the CDN dynamically chooses a server to route the request to, usually one that is appropriately close to the client. Note that this dynamic CDN request routing is an extra step that is not necessary for stand-alone Web servers. Efficient CDN server selection allows CDNs to overcome the extra overhead of the dynamic routing step by taking advantage of improved connectivity to the end user. CDN server selection applies for both static and dynamic content. In the latter case, content can be dynamically assembled at the edge servers [1].

CDNs typically perform dynamic request routing using the Internet's Domain Name System (DNS) [11]. The DNS is a distributed directory whose primary role is to map fully qualified domain names (FQDNs) to IP addresses. To determine an FQDN's address, a DNS client sends a request to its local DNS server. The local DNS server resolves the request on behalf of the client by querying a set of authoritative DNS servers. When the local DNS server receives an answer to its request, it sends the result to the DNS client and caches it for future queries. Each DNS record has a time-to-live (TTL) field that tells the local DNS server how long it may cache the result.

Normally, an authoritative DNS server's association from FQDNs to IP addresses is static. However, CDNs

\*Zhuoqing Morley Mao (email: [zmao@cs.berkeley.edu](mailto:zmao@cs.berkeley.edu)) is a Computer Science graduate student at University of California, Berkeley. This work was done during her internship at AT&T Research Labs.

†Current affiliation: IBM Research

use modified authoritative DNS servers for CDN server selection. The results of a DNS query to one of these DNS servers may vary dynamically depending on factors such as the source of the request and the condition of the network. Typically, the CDN's authoritative DNS server maps the client's local DNS server address to a geographic region within a particular network and combines that with network and server load information to perform CDN server selection. To enable fast reaction to dynamic resource changes, the answer returned by the CDN's DNS server has a small TTL. This approach is largely transparent to the client, and works for any Web content (including both HTML and streaming media).

Although DNS-based server selection is transparent and general, it has two inherent limitations [15, 4]. First, it is based on the implicit assumption that clients are close to their local DNS servers. The CDN DNS server performing dynamic request routing only has access to the client's local DNS server's IP address—it does not know the client's own IP address. However, the assumption that clients are close to their local DNS server may not be valid. For example, the client might be using a local DNS server hierarchy in which the outermost local DNS server that communicates with authoritative DNS servers may be far removed from clients; the client may have been configured with a local DNS server which is far away; or the client may be using a secondary local DNS server that is more distant from it than its primary local DNS server. Therefore, using only the local DNS server information to select CDN servers has the inherent risk of selecting a server farther away from the client than other available CDN servers.

The second inherent limitation of DNS-based server selection is that a single request from a local DNS server can represent differing numbers of Web clients — this is called the *hidden load factor* [8]. The hidden load has implications on a CDN's load balancing algorithm. For example, a DNS request from a local DNS server of a large ISP may result in many more Web requests than a DNS request from a local DNS server of a small site. CDNs need to be able to properly weigh individual DNS requests to distribute Web requests among its CDN servers. If the hidden load factors are known, load balancing algorithms described by Colajanni, et al. [7, 8] can be easily deployed to achieve better load distribution. On the other hand, if the hidden load factors are not known, fine-grained request distribution may be difficult.

We study the extent of the first limitation and its impact on CDN server selection. To this end, we developed a simple, non-intrusive, and efficient mapping technique

to determine the associations between clients and local DNS servers. We deployed this technique on several sites to collect an extensive data set which we use to study the impact of proximity on DNS-based server selection using four different proximity metrics. We conclude that DNS is good for very coarse-grained server selection, since 64% of the associations belong to the same Autonomous System (AS). DNS is less useful for finer-grained server selection, since only 16% of clients use DNS servers in the same *network-aware cluster* [13] (based on BGP routing information). We also measure the CDN server distribution of several real-world CDNs to evaluate whether the proximity of a client to its local DNS server leads to potentially suboptimal CDN server selection decisions in practice. Our technique could also be used to determine hidden load factors by associating the HTTP request pattern in the Web server logs with the DNS request information.

Our work makes the following contributions. We developed a novel measurement methodology and architecture for accurately collecting local DNS server IP addresses of Web clients. We demonstrated its successful deployment on several sites including a large commercial site and through the collection of a huge database of associations. Based on this data, we did an extensive analysis of the proximity between clients and their local DNS servers and discovered that significant improvement in proximity is possible by configuring clients to use a closer local DNS server. Finally, we evaluated the impact of the proximity between clients and their local DNS servers on server selection in three of the largest commercially deployed CDNs. We conclude that DNS is good for very coarse-grained server selection, but less suitable for fine-grained request distribution.

The rest of the paper is organized as follows. Section 2 describes our methodology and measurement setup for gathering DNS client associations. In Section 3, the association results are analyzed in detail to evaluate the proximity between the client and its local DNS server. Then, in Section 4 we study the impact of proximity evaluation on DNS-based server selection in three of the largest commercially deployed CDNs. Related work is covered in Section 5. In section 6, we discuss future work. Section 7 concludes.

## 2 Experimental methodology

In this section we describe our novel technique for determining a Web client's local DNS server. This is a necessary first step in measuring the closeness of clients to their local DNS servers. We also evaluate the impact

of our technique on end user performance. Later, in Section 5, we will explain how our technique is a significant improvement over related previous work in terms of efficiency, nonintrusiveness, and accuracy.

## 2.1 Measurement setup

There are three main components necessary to use our technique: a specialized authoritative DNS server, an HTTP redirector, and a one-pixel embedded transparent GIF image. To obtain a client population we solicited volunteer Web sites. All the volunteers had to do to participate in our study was to add a link to our one-pixel transparent GIF to the end of one or more of their commonly accessed Web pages. Assuming the experiment is hosted by us at `example.com`, this involves adding the following HTML code towards the end of a web page:

```

```

To allow us to easily account for hits from different sites, each participant replaces `xxx` in the URL with a site identifier<sup>1</sup>. This allows us to easily add additional volunteer sites without having to make any changes to our Web or DNS server configuration.

When a Web client loads the one-pixel embedded image, our technique allows us to match the address of the local DNS server resolving host names on behalf of the client with the address of the client itself. This process is shown in Figure 1. First, the client attempts to get the image from `xxx.rd.example.com` — our HTTP redirector. Rather than serving the image, the redirector determines the client's IP address and issues an HTTP redirect to `ipCLI.cs.example.com`, where `CLI` is replaced with a string encoding the IP address of the client (step 2). Next, the client contacts its local DNS server to resolve this domain name (step 3). The client's local DNS server attempts to resolve `ipCLI.cs.example.com` by sending a DNS request to our authoritative DNS server (step 4). At this point our authoritative DNS server logs the IP address of the local DNS server and the client IP address embedded within the query. It then sends the address of the content server hosting the image back to the client's local DNS server (step 5). This resolution is passed on to the client (step 6), which retrieves the image from the content server (steps 7 and 8).

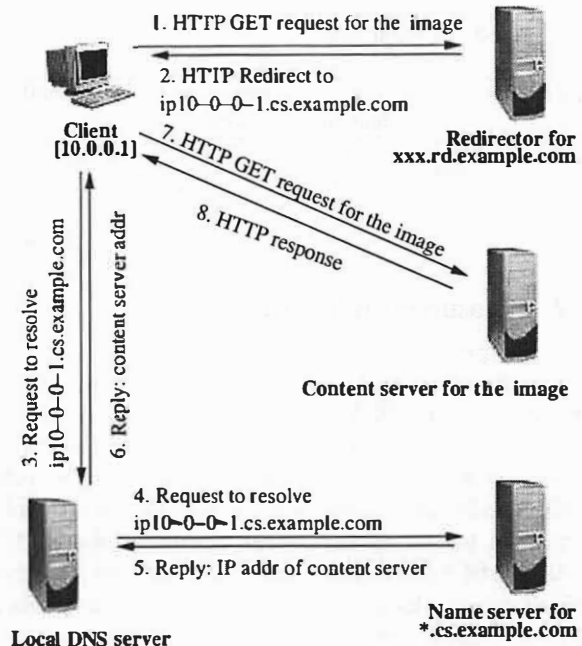


Figure 1: Embedded image request sequence

This measurement methodology has a limitation for clients that do not fetch inlined images and those that abort the page download process before the DNS resolution is made for the embedded image. In these cases, we are unable to collect their local DNS server information.

Note that in some cases, a local DNS server *hierarchy* may exist. The local DNS server recorded in our measurement is the outermost local DNS server which directly contacts the authoritative DNS server for the `example.com` domain. In DNS-based server selection, the CDN's DNS server only sees the outermost local DNS server. In this study, this outermost DNS server is what we refer to as the "local DNS server."

This measurement approach is fully deterministic. It collects one association each time a new client visits a site with the embedded image. Multiple pages on the same site, or subsequent visits to the same page, may result in repeated retrievals of the calibrating image depending on the client's caching policy.

Note that the redirector also logs client requests — this information can be correlated with the DNS and web server logs to obtain the hidden load factors. Statistics on client browsing characteristics can also be gathered from the HTTP headers in the redirector log.

<sup>1</sup>Our authoritative DNS server [6] allows host names to be wildcarded, so we can set an address for `*.rd.example.com`.



Table 1: Keynote image overhead measurements

Location	Avg download latency (sec)		Increased overhead
	without image	with image	
World wide	1.17	1.31	12%
US	1.04	1.14	10%

## 2.2 Measurement impact

Because we propose to use our measurement infrastructure on a production Web site, it is important to evaluate its impact on the server performance and other aspects of its operation. The additional overhead our measurement technique imposes on Web client performance is the retrieval of the transparent image, including the HTTP redirect and extra DNS requests. Because the image is transparent, it does not visually affect the page. Furthermore, the image is small in size—43 bytes—which keeps the added delay to a minimum. We also encourage participants to include the image at the end of the HTML page containing it; therefore, browsers will normally request it last. Thus, the extra latency associated with the image is usually hidden from the user's Web browsing experience. Another advantage of the small size of the image is that when the image is not available for download, it does not affect the visual appearance of the Web page at all.

Our custom HTTP redirector is a single-threaded, non-blocking, 300-line C program. The redirector responds to all Web requests with a "302 Moved Temporarily" HTTP redirect to a URL with the client's IP address embedded in it. Due to the small size and overhead of the redirector, we found it to be highly reliable and more responsive than a standard Web server.

To validate the claim of a small increase in latency, we measured a simple Web page with Keynote [2] to compare the download time with and without the embedded calibrating image. Keynote probes are located in 25 cities within the US and 10 cities outside the US. The Web page we measured had a total size of 39 Kbytes including 13 images and was accelerated by a CDN. The increased overhead percentage is therefore higher than we would expect for a regular unaccelerated Web page with more embedded images. Table 1 shows that the increased overhead averages less than 140 ms, which is 10–12% of the total download time.

We also tested our system to see what would happen in the event of a failure of the redirector, image content server, or DNS server. We found that the impact

Table 2: Participating sites in the study

Site	Type	# of 1-pixel image hits	Duration
1	att.com	20,816,927	2 months
2,3	Personal pages (commercial domain)	1,743	3 months
4	Research lab	212,814	3 months
5-7	University sites	4,367,076	3 months
8-19	Personal pages (university domain)	26,563	3 months

Table 3: DNS and HTTP log statistics for all sites

Type	Count
Client-LDNS associations	4,253,157
HTTP requests	25,425,123
Unique client IPs	3,234,449
Unique LDNS IPs	157,633
Client-LDNS associations where client and LDNS have the same IP address	56,086

of failure on the user is minimal. We tested the failure of these three components using Microsoft Internet Explorer (MSIE) 6 and Netscape Navigator 6 and found that those browsers will first load the rest of the Web page and then time out while trying to fetch the image.<sup>2</sup> There is no visible change to the Web page or any pop-up error message; however, the Netscape logo or MSIE browser logo will provide visual feedback until the browser times out.

## 3 Analysis results

We conducted our measurement study for about three months, and nineteen Web sites participated, as described in Table 2. We classify these sites into two categories: *commercial* (sites 1-3) and *educational* (sites 4-19). As we show in Section 3.1, the client and local DNS associations visiting these two sites have very different characteristics. For ease of discussion, we use *LDNS* to represent a local DNS server. A total of 4,253,157 unique client and LDNS associations were collected. Table 3 presents the statistics of the DNS server and the redirector log for all sites.

To study the proximity between the client and its local

<sup>2</sup>We tested with the default setting without any special options. Some older versions of both browsers were also tested giving the same behavior.

DNS server, we use the following four metrics.

- **AS clustering.** Autonomous System (AS) clustering refers to observing whether a client is in the same AS as its local DNS server. An AS is a region under a single administrative control. A single AS might contain an entire backbone or a large corporation which might span multiple continents. Therefore, AS-based clustering is the most coarse-grained metric we use.

- **Network clustering.** This metric observes whether a client is in the same *network-aware cluster* (NAC) as its local DNS server, where network clusters are identified by the *network-aware clustering* technique [13] using prefix entries from BGP routing table snapshots from a wide set of routing tables. *Longest prefix matching* is used to map clients to network clusters identified by a network prefix. All the clients within a network cluster are topologically close together and with a high probability belong to the same administrative domain. Validation tests (in [13]) using *nslookup* and *traceroute* show that the accuracy of network clustering is above 90% across all the Web logs from the study by Krishnamurthy and Wang. Network clustering is much more fine-grained than AS clustering [12].

For both AS and network clustering, BGP prefixes and the association of IP CIDR blocks to ASes were extracted from an extensive set of BGP tables collected on May 27, 2001 from the sources listed by Krishnamurthy and Wang [13] and Telstra Internet [5]. There are a total of more than 440,000 unique routing entries.

- **Traceroute divergence.** This metric, used previously in [15], is based on the length of divergent paths to the client and its local DNS server from a probe point using *traceroute*. It is defined to be the maximum number of disjoint network hops from a probe location to the client and its LDNS.
- **Round-trip time correlation.** This metric, used previously in both [15] and [4], refers to examining the correlation between the message round-trip times from a probe point to the client and its local DNS server.

AS clustering, network clustering, and traceroute divergence are topology-oriented metrics, while round-trip time correlation is a performance-oriented metric. AS and network clustering are passive, requiring no active probing. The other metrics are highly dependent on the

Table 4: Aggregate statistics of AS/network clustering

Metrics	# of client clusters	# of LDNS clusters	total # of clusters
AS clustering	9,215	8,590	9,570
Network clustering	98,001	53,321	104,950

probe locations. To obtain an exhaustive evaluation of proximity, we include all four metrics in our study.

### 3.1 AS and network clustering

Table 4 shows the aggregate statistics from the data we collected—the number of clusters containing clients, the number of clusters containing local DNS servers, and the total number of clusters. We note that from daily routing table analysis from several major ISPs [9], up to 12,000 unique ASes were identified as being in use on November 12, 2001. The theoretical limit on the possible number of ASes is determined by the 16-bit AS identifier, resulting in a total of 64K ASes. Thus, we observed close to 80% of ASes that were identified on November 12, 2001 and close to 15% of the total possible ASes. With regard to network clusters, the maximum number of network clusters is 440K, since we used 440K unique prefixes. A one day extract from the 1998 Winter Olympic Games server log has 9,853 client clusters [13]. Thus, our measurement data contains close to ten times as many client clusters from one day of a popular Web server log and close to 25% of all possible network clusters. We conclude that the data we collected is extensive and covers a significant number of ASes and network clusters.

Table 5 shows the percentage of client-LDNS associations sharing the same cluster for clients visiting educational sites, commercial sites, and all sites in our measurement study. We observe that clients visiting educational sites have better proximity to their local DNS servers using the network- and AS- clustering metrics. This is expected since most of these clients also come from universities, which generally have a denser distribution of local DNS servers and better local DNS configurations than commercial ISPs. Because the majority of our log results from hits to the commercial sites, the proximity values for clients visiting all participating sites are very close to those visiting commercial sites alone. Because CDNs are most likely to accelerate commercial sites, we believe our client mix is representative

Table 5: Percentage of client-LDNS associations sharing the same cluster classified according to the types of domains visited by the clients

Metrics	Client IPs			HTTP requests		
	educational	commercial	combined	educational	commercial	combined
AS cluster	70%	63%	64%	83%	68%	69%
Network cluster	28%	16%	16%	44%	23%	24%

of clients visiting a CDN-accelerated site. In the following discussion, we consider clients visiting all participating sites.

Using AS clustering, 64% of distinct client-LDNS associations share the same AS. Thus, more than half of the clients use a local DNS server in the same AS. This is expected, since it is common for an administrative domain to run its own DNS server. If users configure their DNS settings correctly, they typically use the LDNS in their administrative domain by default. About 69% of the HTTP requests come from clients using an LDNS server in the same AS cluster. This means clients with LDNS in the same AS are slightly more active than those that use an LDNS in another AS.

The above results indicate that in about 64% of the cases, CDNs could select appropriate servers using DNS redirection with the granularity of ASes. Thus, even if a CDN deployed a cache in every AS in the world, it could select the closest cache according to the AS metric only in 64% of the cases. However, AS clustering does not reveal how well redirection works for finer-grained load-balancing. An AS can span large geographical regions, causing network delays between two hosts within the same AS to be relatively high. For finer-grained load-balancing it is therefore important to consider network clustering, which groups together IP addresses that are close together topologically and likely to be under the same administrative domain.

The observations using network clustering are significantly different from the AS clustering results. Only 16% of the client-LDNS associations are in the same network cluster. This shows that most clients are *not* in the same routing entity as their local DNS servers. If the HTTP request count is taken into account, about 24% of the HTTP requests in our logs originated from clients that used an LDNS in the same network cluster. Again, the difference between these two numbers demonstrate that clients with LDNS in the same network clusters are more active than those with LDNS in a different network cluster.

Overall, these results indicate that DNS-based redirection can confidently select appropriate CDN servers with the granularity of an AS. However, for CDNs with multiple servers in the same AS, the selection may not be as accurate. If there is a CDN server in each network cluster, then DNS-based redirection will only select the CDN server in the same network cluster as the client about 24% of the time.

### 3.2 Traceroute divergence

Another metric to evaluate the proximity between the client and its local DNS server is the maximum number of disjoint network hops from a probe location to the client and its local DNS server. In [15], this metric is referred to as the *traceroute cluster size*. The smaller the cluster size or *traceroute divergence*, the closer the client is to the local DNS server. In many of our traceroute results, we found that the network routes from the probe site to the client and its LDNS diverge and converge multiple times due to router load balancing. We use the last point of divergence as the reference for calculating disjoint network hops. For example, Table 6 shows the network routes obtained by performing traceroute to the client 112.74.197.163<sup>3</sup> and its LDNS 112.25.195.1. We use hop 11 instead of 2 as the point of divergence. Thus, the traceroute divergence in this example is  $\max(14 - 11, 13 - 11) = 3$ .

We selected four probe sites representing candidate CDN servers and performed traceroute to a sample of clients and local DNS servers from the log. The sample consists of 48,908 client-LDNS pairs or 66,975 IP addresses. It is obtained by randomly selecting one client-LDNS pair from the top half of the client network clusters generating the most HTTP requests. The number of client-LDNS pairs reached by an individual probe site ranges from 9,878 to 11,935. In about 20% of these, both the client and the LDNS belong to the same network cluster. And in about 75% of these, both the client and the LDNS belong to the same AS cluster.

<sup>3</sup>For privacy concerns, the IP addresses have been anonymized.

Table 6: Traceroute divergence

1 112.0.1.1 6 ms	1 112.0.1.1 5 ms
2 112.124.182.17 6 ms	2 112.124.182.17 15 ms
3 112.123.1.107 ms	3 112.123.1.22 14 ms
4 112.122.1.149 8 ms	4 112.122.5.246 7 ms
5 112.122.2.173 25 ms	5 112.122.2.2 24 ms
6 112.122.2.206 32 ms	6 112.122.2.206 31 ms
7 112.122.2.41 34 ms	7 112.122.2.41 35 ms
8 112.122.2.26 71 ms	8 112.122.2.26 68 ms
9 112.122.2.121 75 ms	9 112.122.2.121 77 ms
10 112.123.145.25 73 ms	10 112.123.145.25 72 ms
11 112.124.23.6 72 ms	11 112.124.23.6 73 ms
12 112.25.192.2 72 ms	12 * * *
13 112.25.192.181 73 ms	13 * 112.25.195.1 71 ms
14 112.74.197.163 92 ms	

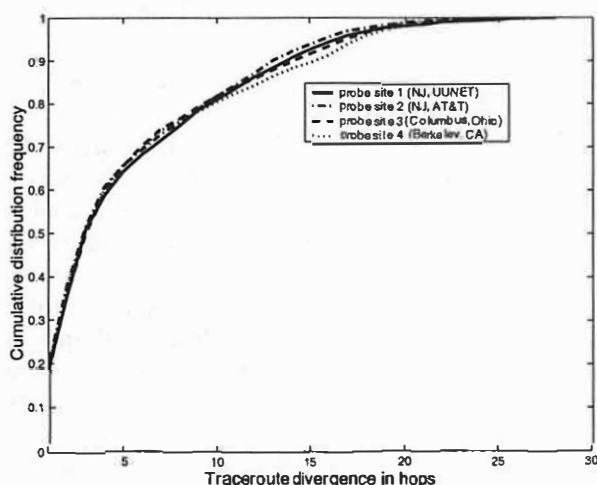


Figure 2: Proximity evaluation using traceroute divergence

Figure 2 shows the cumulative distribution of traceroute divergence for the sampled client-LDNS pairs. About 14% of them have traceroute divergence of 1. The mean divergence varies from 5.8 to 6.2 depending on the probe site, and the median traceroute divergence is 4 from all four probe sites. This means that a large fraction of clients are topologically quite *close* to their local DNS servers using the hop count metric. At most 30% of the client-LDNS pairs have traceroute divergence of size 8. This result is slightly inconsistent with the results described by Shaikh, et al. [15] considering 1,090 client-LDNS pairs of dial-up ISPs. We believe that the difference can be explained by the fact that our results are based on the analysis of a much larger set of populations visiting both commercial and educational sites.

The absolute values of traceroute divergence may not be completely indicative of the proximity of a client to its

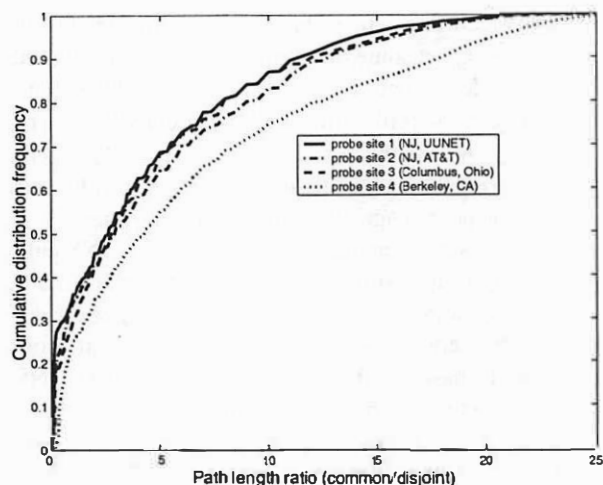


Figure 3: Ratio of common to disjoint path length

local DNS server. In Figure 3, we plot the ratio of the common path length to the disjoint path length from a probe site. Using the terminology of Shaikh, et al. [15], the common path length is the minimum number of network hops of the shared path from the probe site to the local DNS server and the client before their paths diverge. For example, the common path length of client 112.74.197.163 and its LDNS 112.25.195.1 (shown in Table 6) is  $\min(11, 11) = 11$ . The disjoint path length is the maximum number of network hops of the diverging paths. In this example, the divergent path length is  $\max(14-11, 13-11)=3$ . Again, we use the last point of the divergence as the reference point. For all probe sites, less than 34% of the client-LDNS pairs have disjoint paths at least as long as the common path. This means that at least 66% of client-LDNS pairs have a common path as long as or longer than their disjoint path. This metric implies that most clients are topologically close to their LDNS as viewed from a randomly chosen probe site.

### 3.3 Round-trip time correlation

Some CDNs select servers based on the round-trip latency between the CDN server and the client's local DNS server [15]. It is therefore important to understand the correlation between the round-trip delay to a client and to its LDNS from a third location.

To compare with the results presented in [15], we study how the round-trip delays to the client and its LDNS determine the accuracy of the CDN server selection based on round-trip delays to the LDNS. Since our data set consists of more than 4.2 million pairs of client and

LDNS, much larger than that presented in [15] (1,090 pairs), we expect some differences. Let  $t_c^i$  and  $t_d^i$  be the round-trip delays between the probe site  $i$  and the client, and between the probe site  $i$  and the client's LDNS, respectively. We ask the question whether  $t_d^i < t_c^i$  implies  $t_c^j < t_d^j$ . Depending on the locations of two probe sites  $i$  and  $j$ , the percentage of violations ranges from 17% to 38%. For instance, among the 9,360 client-LDNS pairs responding to traceroute from both probe site 1 and 2, about 38% violate this assumption. This implies that if one selects between two CDN servers located at probe sites 1 and 2 based on the round-trip delays to the LDNS, the decisions would be suboptimal 38% of the time for the set of clients considered based on the round-trip delay metric. On the other hand, among the 7,895 pairs responding to traceroute from both probe site 2 and 4, only 17% violate this assumption. This means that this metric is highly dependent on probe locations. However, it is a reasonable metric for use to avoid really distant servers.

Another interesting question to answer is whether, if two CDN servers are roughly an equal distance from the LDNS based on the round-trip delay, the same holds from the client's perspective. Thus, we ask whether  $|t_d^i - t_d^j| \leq w$  implies  $|t_c^i - t_c^j| \leq w$ , where  $w$  is a small number (e.g., a 10 ms threshold was used by Shaikh et al. [15]). In the sample of our study, it holds in 44–75% of the cases depending on the probe sites. This number is bigger than the previously obtained result of 12% in [15].

### 3.4 Improved local DNS configuration

For the client and local DNS associations that are not in the same network cluster, we ask whether there exist any local DNS servers in those clusters. From our log, we collected a set of local DNS servers. Thus, assuming the clients have access to those local DNS servers in their network clusters, it is interesting to examine the degree of improvement if all LDNS servers were used optimally. This assumption is not unreasonable, since most IP addresses in the same network cluster are under the same administrative control. From Table 4, we can calculate the number of client ASes and network clusters where there are no local DNS servers as observed in our log. There are  $9,570 - 8,590 = 980$  such AS clusters, and  $104,950 - 53,321 = 51,629$  such network clusters. Table 7 compares the improved percentages of client-LDNS associations and HTTP requests in the same cluster with the original results. If the clients in our data currently configured to use a LDNS in a different cluster are allowed to use an LDNS in the same cluster, then at least 92% of the HTTP requests come from clients using the

Table 7: Improvement of the percentage of the client-LDNS associations sharing the same cluster using optimal LDNS assignment

Metrics	Client IPs		HTTP requests	
	Original	Improved	Original	Improved
AS cluster	64%	88%	69%	92%
Network cluster	16%	66%	24%	70%

LDNS in the same AS cluster. That number is 70% for network clusters.

### 3.5 Clients using multiple local DNS servers

Some client IP addresses in our data are associated with multiple LDNS IP addresses. This may happen due to the following reasons: (1) The first LDNS server the client contacts times out and the second LDNS server is contacted. (2) The client's LDNS server is configured by a DHCP server that assigns the LDNS server IP addresses from a set of addresses in a round-robin fashion. (3) A client may be configured to round-robin among multiple LDNS servers. (4) The client IP address is reused at different times by different users and these users may have different configurations for their LDNS servers, resulting in different associations. (5) The client IP address is that of a NAT box or a application-level proxy, so there are multiple actual clients behind this IP address using different LDNS servers. (6) The client is misconfigured.

Here we examine the distribution of the LDNS servers with which a client IP address is associated. If they all occupy the same cluster as the client, DNS-based server selection can use the local DNS server's IP address to estimate where the client is even if the client uses multiple local DNS servers. However, if they occupy multiple clusters or a single cluster different from the client, it is more difficult to use DNS-based server selection. In Table 8, we show how many clients use ten or fewer local DNS servers. In addition, we observe that some IP addresses are associated with up to 330 local DNS servers occupying up to 273 different network clusters. Further investigation shows that some of these addresses belong to cache proxies. In general, we observe that the more LDNS servers with which a client IP address is associated, the lower the percentage of associations with the client and LDNS in the same cluster. Fortunately, the majority of client IP addresses are associated with a single LDNS server. They are responsible for about 52% of the requests. However, only about 20% in this group



Table 8: Clients using ten or fewer multiple local DNS servers

# of clients (% of total)	# of LDNS (avg # of NACs)	% of total HTTP requests	% associations with client and LDNS in the same NAC
2,524,939 (78.064)	1 (1.0)	51.8	20.3
522,228 (16.146)	2 (1.6)	22.4	12.1
123,524 (3.819)	3 (2.1)	10.4	6.6
41,422 (1.281)	4 (2.5)	4.9	4.7
13,469 (0.416)	5 (2.9)	2.5	4.9
4,555 (0.141)	6 (3.3)	1.8	6.7
1,590 (0.049)	7 (4.1)	1.3	9.9
713 (0.022)	8 (4.7)	0.7	13.6
461 (0.014)	9 (5.5)	0.7	14.2
273 (0.008)	10 (6.1)	0.5	14.0

have the client and LDNS in the same network cluster.

### 3.6 Comparisons of proximity metrics

Given the above set of metrics for evaluating proximity between client and its local DNS server, we compare their results on a common set of 7,894<sup>4</sup> client-LDNS associations in Table 9. The comparison shows that network clustering is a fine-grained metric, similar to trace-route divergence (TD) count of 1. Hosts within the same network cluster, or which have a TD of 1, are guaranteed to be very close to each other. However, hosts not in the same network cluster, or have a TD bigger than 1, may still be quite close. Thus, these two metrics are quite conservative. AS clustering is the most coarse-grained metric, since an AS can be quite large. This is comparable to the ratio of common to disjoint path length. RTT correlation is also a relatively coarse-grained metric. It is inconclusive and largely dependent on the two probe site locations.

In general, performance-oriented metrics such as round-trip time should provide accurate real-time network latency measurements. CDNs often do real-time network measurements from their servers to clients. Since we can only probe from a limited set of locations, such metrics are inconclusive. Topology-oriented metrics have the advantage of being non-invasive, since they do not incur any network overhead. However, they cannot take network congestion into account.

As we explain in the following section, the applicability of each metric depends on the density of CDN server placement. The denser the placement, the more fine-

<sup>4</sup>Only 7,894 of all associations can be reached from both probe sites 2 and 3.

Table 9: Comparison of four proximity metrics

Proximity metric	Evaluation
AS clustering	78% in the same cluster
Network clustering	23% in the same cluster
Traceroute divergence (TD) (probe site 2)	16%: TD=1, 32%: TD=2 median TD=4, mean TD =5.7 65%: $disjointPathLen \leq commonPathLen$
RTT correlation (probe sites 2, 3)	71%: $t_d^2 < t_d^3 \Rightarrow t_c^2 < t_c^3$ 62%: $ t_d^2 - t_d^3  \leq 10ms \Rightarrow$ $ t_c^2 - t_c^3  \leq 10ms$ $a = t_d^2 - t_d^3, b = t_c^2 - t_c^3$ $correl(a,b) = 0.13$

grained metric is needed.

## 4 Application impact

In this section, we focus on the impact that client-LDNS associations have on DNS-based server selection. We study this impact in detail for three of the largest commercial CDNs. We anonymize the CDN names to properly reflect the nature of this work as a research vehicle rather than any form of competitive analysis. All three CDNs chosen rely on deploying caches in multiple networks. ISP-based CDNs deployed by companies like AT&T and Qwest are excluded from this study, since their caches are located in one or two ASes. Since a client and its LDNS are very likely to be in the same AS (about 69% of HTTP requests in our study), an ISP-based CDN can easily identify a peering link that is suitable for the AS containing both of them<sup>5</sup>. The results described below are representative of all the data we collected and remained stable during our entire study.

Previous work by Johnson, et al. [10] has shown that DNS-based CDNs do not always pick the best server available. Here we study whether this is partly due to the inherent limitations of DNS-based server selection. The answer to this largely depends on the proximity between clients and local DNS servers and the location of CDN servers.

The proximity evaluation of client-LDNS associations using the network clustering metric indicates that, if a CDN had a server in each network cluster, about 84% of the selection decisions for the client population in our log could be *suboptimal*. This is because our study

<sup>5</sup>The main tradeoff here is fewer peering links traversed in multi-ISP CDNs versus less traffic between access and backbone routers as well as lower costs in single-ISP CDNs.

found only 16% of these clients have their LDNS in the same network cluster. For clients with their LDNS in different network clusters, the CDN would most likely resolve the DNS query from a client's LDNS to the CDN server in the LDNS's cluster and not the cluster where the client resides. In reality, and as we show below, even the biggest CDN today does not have a CDN server in every network cluster. Thus, it is important to examine the impact of DNS-based redirection in a commercial content distribution setting.

We assume that on average a CDN server within the client's AS/network cluster or smaller traceroute divergence (TD) is closer than one in a different cluster or larger TD. For clients with CDN servers in their clusters, if a CDN selects a server not in a client's cluster, this may be a suboptimal decision in terms of proximity. We also assume that CDNs attempt to optimize for proximity in most cases. Network bandwidth is less important, since the content delivered by these CDNs is relatively small in size. Although CDNs may also incorporate the avoidance of overloaded servers in their server selection algorithms, we believe that our assumption is reasonable because CDNs today are highly overprovisioned from the perspective of server capacity. Furthermore, we repeated our experiments on separate dates to avoid any possibility of a skew due to a flash event, and the results were always similar. One limitation in our results below is that we do not quantify suboptimal server selection in terms of end user performance, nor how close it is to the optimal server selection.

We first describe our measurement methodology then use AS/network clustering and traceroute divergence to study how the proximity between client and LDNS affect DNS-based server selection in three commercial CDNs.

## 4.1 Experiment methodology

We use the following three data sets for our study.

1. **Client-LDNS associations.** These associations between clients and their LDNS servers are obtained from our measurement study.
2. **LDNS-CDN server associations.** For a given CDN, these associations map LDNS servers from the first data set to the CDN servers selected by the CDN when resolving a query from these LDNS servers.
3. **Available CDN servers.** This data set represents a list of CDN servers available in a given CDN.

In the first data set, we sampled 42,991 LDNS servers from our measurement study. We obtained the second data set by sending DNS queries to these 42,991 LDNS servers using the *dig* command for a domain name of a Web site that we know is a customer of a given CDN. 27,918 of these LDNS servers do not use access control and hence answered the queries from our machines, as if these machines were their clients. To answer our queries, these LDNSs recursively resolved our queries with the CDN in question. The server selected by the CDN for this DNS query is exactly the same server that would be used by any real client associated with this LDNS, as if that client and not our machine initiated the DNS query.<sup>6</sup>

The third data set was obtained in a similar way, except we added a large number of additional LDNS servers to the 27,918 LDNS servers above, for a total of 41,754 different local DNS servers. This is to increase the likelihood of finding all CDN servers of a particular CDN for a given domain. The extensive list of geographically distributed LDNS servers was obtained from DNS server logs for a large Web site. The set of servers to which a given CDN resolved queries from these LDNSs represents the servers available in this CDN at the time of the experiment. We obtained our second and third data sets at around the same time each day to find the set of servers available to a CDN at the time it performed its server selection in the second experiment.

Note that our set of available servers is conservative, since we might not have discovered all available CDN servers. However, if a CDN performs a suboptimal server selection among a subset of all available servers, its server selection will remain suboptimal for a larger set: suboptimal means that we already found a *closer* server to the client than the one selected by the CDN. A superset of the list of servers would suffer from the same suboptimal assignment.

Many CDNs claim a much larger number of caches. However, CDNs do not utilize all servers for all Web sites and many of their locations may contain multiple caches. The statistics we gathered are for a particular domain served by a CDN. For example, when examining multiple different domain names served by the largest CDN in our study, we found multiple CDN IP address sets of approximately equal size which only partly overlapped. Each unique server IP address we discover may also account for multiple servers.

<sup>6</sup>Note, for fault-tolerance, most CDN DNS servers usually return multiple IP addresses. In this case, we pick the first one, since clients also typically choose the first IP address.

Table 10: CDN cache servers for a particular domain name

CDN	# of AS clusters with servers	# of network clusters with servers	# of CDN servers IPs
CDN X	622	740	1,567
CDN Y	120	152	195
CDN Z	60	79	154

Table 11: The evaluation of server selection according to AS clustering

CDN	CDN X	CDN Y	CDN Z
Clients w/ CDN server in cluster	1,679,515	1,215,372	618,897
Verifiable clients	1,324,022	961,382	516,969
Misdirected clients (% verifiable clients) (% clusters occupied)	809,683 (60%) (92%)	752,822 (77%) (94%)	434,905 (82%) (94%)
MC w/ LDNS not in client's cluster (% misdirected clients)	443,394 (55%)	354,928 (47%)	262,713 (60%)

Table 10 shows the statistics of the CDN server IP addresses of the three CDNs studied for a single domain name obtained on August 7, 2001. These numbers were fairly stable during the course of our study. All three CDNs examined appear to redirect client requests by using DNS, although they may differ in the details of the algorithms. This table lists the total number of CDN servers discovered and the number of AS and network clusters these CDN servers represent. The data in Table 10 confirm our conjecture that CDNs today cover only a small number of all available network clusters for a single domain they serve. While the overall list of LDNSs used for generating the third data set represents 5,788 AS and 21,786 network clusters, the discovered CDN servers represent only a small fraction of these, even in the case of the largest CDN in our study.

With the three data sets above, we evaluate the quality of server selection by these CDNs by examining what percentage of clients are actually redirected to servers in their own cluster, among those clients that have at least one server in their cluster.

Table 12: The evaluation of server selection according to network clustering

CDN	CDN X	CDN Y	CDN Z
Clients w/ CDN server in cluster	264,743	156,507	103,448
Verifiable clients	221,440	132,567	90,264
Misdirected clients (% verifiable clients) (% clusters occupied)	154,198 (68%) (77%)	125,449 (94%) (82%)	87,486 (96%) (93%)
MC w/ LDNS not in client's cluster (% misdirected clients)	145,276 (94%)	116,073 (93%)	84,737 (97%)

## 4.2 Results of DNS-based server selection in commercial CDNs

Tables 11 and 12 show the results of our server selection evaluation using AS and network clustering. We collected 3,234,449 distinct client IP addresses in our logs. The first row of the table contains the number of clients with CDN servers in their clusters for the considered CDNs. Depending on the server density of each CDN, the number of clients with servers in their AS clusters ranges from 19% to 52% of the total clients in the log. This fraction is an order of magnitude lower in the context of network clusters. Thus, according to either metric, most clients will have to be served by *remote* servers. But a more interesting question is how many clients that could have been served by local servers are in reality directed to remote ones.

To answer this question, we concentrate on clients with servers in their clusters and consider the LDNS-CDN server associations for these clients from the second data set. Unfortunately, not all of these LDNS servers respond to DNS queries from our machines. The second row of the tables gives the number of clients, among those with CDN servers in their clusters, whose LDNS servers responded to our queries. We call these clients *verifiable* because we could find out which CDN servers a CDN would redirect these clients to. The third row shows the number of clients that a CDN directed to an external CDN server (one that was outside the client's cluster), when there was an available CDN server within that cluster. We refer to such clients as *misdirected* clients (MC) based on the assumption that CDN servers within the cluster are closer than external ones, although we accept that other factors than proximity may have affected the assignment. We see a large number of misdirected clients according to both proximity metrics. To confirm that these misdirected clients are not due to any

anomaly of clients belonging to a small number of clusters, we also show in the third row the percentage of clusters occupied by these clients relative to the total number of clusters of verified clients. The cluster percentage values are at least as big as the client percentage values. This means that the misdirected clients are fairly spread out in the number of clusters they occupy.

We conjecture that the reason that these clients are misdirected is that their LDNS servers are topologically distant from these clients. CDNs select a server close to the LDNS servers. The servers selected may therefore be suboptimal from the client's perspective. The last row of the tables shows misdirected clients with their LDNS outside their clusters. This row indicates the number of clients that inherently cannot be directed to the most proximal server using a DNS-based mechanism. According to Table 11, for AS clustering, they represent only half of misdirected clients. To understand why CDNs choose a CDN server in a different AS than the one containing the client and its LDNS server, we sampled a dozen of these clients using *traceroute* followed by DNS name resolution of the last-hop router IP address to estimate the geographic locations<sup>7</sup> of the client, CDN servers in the client's AS, and selected CDN servers in a different AS. We found that in most cases, the selected CDN servers by CDNs are geographically closer to the client than CDN servers in the same AS. Assuming peering links between the client's AS and the selected CDN server's AS are not congested, redirecting to a nearby CDN server in a different AS may be a better decision than redirecting to a distant CDN server in the same AS. This observation also confirms our finding that AS clustering is a very coarse-grained metric for evaluating proximity.

For network clustering, the last row of Table 12 indicates that an overwhelmingly *majority* of misdirected clients have their LDNS servers in a different network cluster. This confirms our hypothesis that such misdirection is due to the fact that clients and their LDNS servers are often not proximal. It also shows the usefulness of network clustering because it is a fine-grained metric for evaluating proximity. We emphasize that we do not know the exact server selection policy used by a commercial CDN, so we cannot fully evaluate the effectiveness of its server selection decisions. However, given that there is such a strong correlation between misdirection and an LDNS being in a different cluster, we can infer that when the LDNS and client do not belong to the same network cluster, this limits the accuracy of server selection.

<sup>7</sup>In many cases, the router's DNS name has an indication of the geographic location [14].

Table 13: The evaluation of server selection according to traceroute divergence (TD) from probe site 3

CDN	CDN X	CDN Z
Client-LDNS pairs examined	2,105	2,171
Clients with CDN servers at smaller TD than ones redirected to	1,606 (76%)	1,724 (79%)
Median TD of CDN servers clients redirected to	11	13
Median TD of closest CDN servers to clients	5	9
Median TD improvement	6	4

Table 13 shows the evaluation of DNS-based server selections according to the traceroute divergence metric.<sup>8</sup> We performed traceroute from probe site 3 to a sample of client and local DNS servers from the log and the CDN cache servers from the third data set. The sample is chosen by randomly selecting one client-LDNS pair from the top 1200 client clusters generating the most HTTP requests. We found over 70% of the clients to be directed to a CDN server that is more distant than another available CDN server. Selecting the closest CDN server would have reduced traceroute divergence by as much as 19 hops for some clients.

Overall, we conclude that, among the clients we could verify, knowing the client's IP address would allow more accurate server selections in a large number of cases (443,394 for CDN X). The last row of Tables 11 and 12 also indicate the number of improved CDN server selections if the client's IP address were known to the CDN. Relative to the total number of clients, in the case of CDN X, this represents a small percentage: specifically 14% (443,394 out of 3,234,449). In general, the number of misdirected clients depends on the server density, placement, and selection algorithms.

## 5 Related work

Our work is motivated by a related effort by Shaikh, et al. [15] examining the effectiveness of DNS-based server selection. They developed a method of finding client-LDNS associations using time correlations of DNS and HTTP requests from DNS and Web server logs. However, as they have noted, the associations obtained using their method are inherently inaccurate due to clock skews, client DNS caching, and mishandling of TTLs. To resolve ambiguities, they used heuristics based

<sup>8</sup>We were unable to include CDN Y in the traceroute experiment, since most of its CDN servers are unreachable using traceroute.

on AS numbers and domain names to decide whether a client and a nameserver did in fact belong together. This heuristic removed misconfigured client-nameserver pairs and did not assure the correctness of associations. They also obtained a set of 1090 client-LDNS associations from accounts with 9 commercial ISPs to study the proximity correlations.

In comparison, our method provides accurate associations eliminating any need for validation. Furthermore, our study has more than 4.2 million associations, consisting of clients from a diverse set of ISPs, far exceeding their data set of 1090 associations.

More recently, Bestavros, et al. [4] have also developed a method for finding client-LDNS associations by assigning multiple IP addresses to a Web server and correlating DNS lookups with client IPs based on the server IP used. This method is slow in discovering client-LDNS pairs due to the limited number of IP addresses a Web server can have. In addition, their method is complicated to implement, requiring reassignment of server IPs and modification of the Web server.

Compared to both works, the distinguishing features of our measurement methodology are efficiency, nonintrusiveness, and accuracy. This allowed us to collect more extensive data, which we used to evaluate the effectiveness of DNS-based server selections using four different proximity metrics in several real-world CDN settings. To our knowledge, we are the first to conduct such an exhaustive proximity evaluation between clients and their local DNS servers using such a representative data set. We are also not aware of other work in examining the impact that the proximity between the local DNS server and the client has on DNS based server selection in commercial CDNs.

There has been a recent effort within the IETF to categorize different mechanisms for request routing in CDNs [3]. DNS-based redirection is one of those mechanisms, and our methodology may prove useful in evaluating the effectiveness of this technique in that context.

## 6 Future work

There are three areas of future work we plan to pursue. First, we plan to study the hidden load factors due to differing amounts of HTTP load corresponding to a DNS name resolution request from an LDNS server. With the help of a busy Web site, we will be able to gather statistics on the number of HTTP requests and clients behind each LDNS server. Identifying LDNS servers resulting

in large numbers of HTTP requests is essential for proactive load balancing and flash crowd protection.

Second, we plan to improve existing DNS-based server selection algorithms by considering the properties of known client-LDNS associations for an LDNS that requests a server name resolution. The following characteristics of the associations can be explored based on data collected using our methodology: known client proximity to the LDNS, known client distribution, and hidden load factor.

Given a name resolution request from an LDNS, if the known client proximity to the LDNS is good, then a CDN server close to the LDNS would also be close to its clients. If the proximity correlation is low, known client distribution and client cluster request patterns would be considered. If the majority of HTTP requests belong to a single network cluster, finding a CDN server close to or within that network cluster would also be close to clients issuing a majority of requests. Along with these factors, the hidden load factor of the LDNS is also considered to select lightly loaded CDN servers for an LDNS with a large hidden load factor. If the proximity correlation is low between LDNS and its clients, then server selection is optimized using other metrics such as server load.

Finally, we would like to apply the results of this work to improving content distribution internetworking (CDI), which refers to the interoperability among multiple CDNs for additional flexibility. A prototype of CDI, called *CDN Brokering* [6], uses a DNS-based brokering mechanism to forward requests among DNS servers of the interoperating CDNs. As a third area of future work, we plan to improve CDN brokering algorithms by using hidden load factors and client-LDNS proximity information. The client-LDNS proximity findings in our work justify DNS-based brokering, because the majority of the clients and their LDNS belong to the same AS.

## 7 Conclusion

In this paper, we propose a novel technique for finding client and local DNS server associations and potentially hidden load factors in a fast, non-intrusive, and accurate manner. Based on the results, we evaluate the proximity between clients and their LDNS using four metrics: AS clustering, network clustering, traceroute divergence, and round-trip time correlation.

We evaluate the potential effectiveness of DNS-based server selection in CDNs based on these metrics. We conclude that DNS is good for very coarse-grained



server selection, since 64% of the associations belong to the same AS. DNS is less useful for finer-grained server selection, since only 16% of clients use a DNS server in the same network-aware cluster. These values can be improved to 88% and 66% respectively, if clients are configured to use a closer local DNS server. Since current CDNs are not present in many network-aware clusters, we conclude that although DNS-based server selection has inherent limitations due to potentially poor proximity correlation between a client and its LDNS, the impact is small due to the sparse distribution of CDN servers in today's CDNs.

At least one CDN has stated a goal of ultimately placing CDN servers in every edge network. The high fraction of clients using LDNS servers in different network-aware clusters suggests that CDNs may be unable to use DNS request routing for such fine-grained server selection unless DNS itself scales to provide each edge network with a local DNS server that communicates directly with the Internet. Thus, from an economic perspective, due to the inherent limited precision of DNS-based server selection, it is less beneficial to have so many CDN servers that the performance to two nearby servers is indistinguishable.

In addition to the proximity evaluation and the novel measurement methodology, our work also provides two additional contributions in improving DNS-based CDNs in general. From our observation, client-LDNS associations are fairly static. Thus, CDNs can build up a database of such associations to infer the geographic location of clients associated with an LDNS IP address to improve server selection. Furthermore, based on the URL-rewriting technique in our measurement methodology, CDNs can completely eliminate the originator problem by embedding the client IP addresses in the URLs of the Web pages, when a client initially requests the base page.

## 8 Acknowledgement

We thank all participants in our measurement study. We especially thank Ted Kowalski of AT&T, Danielle Gallo of AT&T Research, Alex Brown and Milan Andric of UC Berkeley, Frans Kaashoek of MIT, and Mike Dahlin of UT Austin for generously offering their main Web pages for our study. We are grateful to Robert Szweczyk and Alec Woo for instrumenting the TinyOS Web page. We also thank Hyunseok Chang, Yuan Gao, Jason Hong, David Oppenheimer, Amit Sehgal, Wilson So, and Hao Zhang, who took part in our measurement using their personal Web pages.

## References

- [1] Edge side includes. <http://www.esi.org>.
- [2] Keynote systems. <http://www.keynote.com/>.
- [3] A. Barbir, B. Cain, F. Douglass, M. Green, M. Hofmann, R. Nair, D. Potter, and O. Spatscheck. Known CN Request-Routing Mechanisms, February 22 2002. Work in Progress, draft-ietf-cdi-known-request-routing-00.txt.
- [4] Azer Bestavros and Sumit Mehrotra. DNS-based internet client clustering and characterization. Technical Report BUCS-TR-2001-012, Boston University, 2001.
- [5] Telstra Internet. <http://www.telstra.net/ops/bgptab.txt>.
- [6] Alex Biliris, Charles D. Cranor, Fred Douglass, Michael Rabinovich, Sandeep Sibal, Oliver Spatscheck, and Walter Sturm. CDN brokering. In *Proceedings of 6th International Workshop on Web Caching and Content Distribution*, June 2001.
- [7] M. Colajanni, P. S. Yu, and D. M. Dias. Analysis of task assignment policies in scalable distributed Web-server systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(6):585–600, 1998.
- [8] Michele Colajanni, Philip S. Yu, and Valeria Cardellini. Dynamic load balancing in geographically distributed heterogeneous web servers. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, May 1998.
- [9] Geoff Huston. Internet bgp table. <http://www.telstra.net/ops/bgp/>, November 2001.
- [10] Kirk L. Johnson, John F. Carr, Mark S. Day, and M. Frans Kaashoek. The measured performance of content distribution networks. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, 2000.
- [11] B. Krishnamurthy, C. Wills, and Y. Zhang. On the use and performance of content distribution networks. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop (IMW'2001)*.
- [12] Balachander Krishnamurthy and Jia Wang. Topology modeling via cluster graphs. *Proceedings of SIGCOMM IMW 2001*.
- [13] Balachander Krishnamurthy and Jia Wang. On Network-Aware Clustering of Web Clients. In *Proceedings of ACM SIGCOMM'2000*, 2000.
- [14] Venkata N. Padmanabhan and Lakshminarayanan Subramanian. An investigation of geographic mapping techniques for internet hosts. In *Proceedings of the ACM SIGCOMM 2001*.
- [15] Anees Shaikh, Renu Tewari, and Mukesh Agrawal. On the effectiveness of DNS-based server selection. In *Proceedings of IEEE Infocom 2001*, 2001.

# Geographic Properties of Internet Routing

Lakshminarayanan Subramanian  
University of California, Berkeley  
lakme@cs.berkeley.edu

Venkata N. Padmanabhan  
Microsoft Research  
padmanab@microsoft.com

Randy H. Katz  
University of California, Berkeley  
randy@cs.berkeley.edu

## Abstract

In this paper, we study the geographic properties of Internet routing. Our work is distinguished from most previous studies of Internet routing in that we consider the geographic path traversed by packets, not just the network path. We examine several geographic properties including the circuitousness of Internet routes, how multiple ISPs along an end-to-end path share the burden of routing packets, and the geographic fault tolerance of ISP networks. We evaluate these properties using extensive network measurements gathered from a geographically diverse set of probe points. Our analysis shows that circuitousness of Internet paths depends on the geographic and network locations of the end-hosts, and tends to be greater when paths traverse multiple ISP. Using geographic information, we quantify the degree to which an ISP's routing policy resembles hot-potato or cold-potato routing. We find evidence of certain tier-1 ISPs exhibiting hot-potato routing. Finally, based on network topology information gathered at CAIDA, we find that many tier-1 ISP networks may have poor tolerance to the failure of a single, critical geographic node, assuming the published topology information is reasonably complete.

## 1 Introduction

The Internet consists of several autonomous systems (ASes) that are under the control of different administrative domains. Routing across these administrative domains is accomplished using the Border gateway protocol (BGP), a protocol for propagating routes between ASes. ASes connect to each other either at public exchanges or at private peering points. The network path between two end-hosts typically traverses multiple ASes. BGP is flexible in allowing each AS to apply its own local preferences, and export and import policies for route selection and propagation. The characteristics of an end-to-end path are very much dependent on the policies employed by the intervening ASes.

Previous work on Internet routing has focused on studying properties such as end-to-end performance, routing stability, and routing convergence that are affected by routing policies. There has also been work on strategies

for determining alternate (and hopefully better) routes by using overlay networks to circumvent the default Internet routing. We discuss previous work in more detail in Section 2.

In this paper, we present a novel way of analyzing certain properties of Internet routing. We show how *geographic* information can provide insights into the structure and functioning of the Internet, including the interactions between different autonomous systems. In particular, geographic information can be used to quantify well-known network properties such as hot-potato routing. It can also be used to quantify and substantiate prevalent intuitions about Internet routing, such as the relative optimality of intra-ISP routing compared to inter-ISP routing.

To analyze geographic properties of routing, it is necessary to first determine the *geographic* path of an IP route. The geographic path is obtained by stringing together the geographic locations of the nodes (i.e., routers) along the network path between two hosts. For instance, the geographic path from a host in Berkeley to one in Harvard may look as follows: Berkeley  $\rightarrow$  San Francisco  $\rightarrow$  New York  $\rightarrow$  Boston  $\rightarrow$  Cambridge. The level of detail in the geographic path would depend on how precisely we are able to determine the locations of the intermediate routers in the path. In Section 3, we describe GeoTrack [13], a tool we have developed for determining the geographic path of routes. Our study is based on extensive traceroute data gathered from 20 hosts distributed across the U.S. and Europe and also traceroute data gathered by Paxson [26] in 1995.

Internet routes can be highly circuitous. For instance, we observed a route from a host in St. Louis to one in Indiana (328 km away) that traverses a total distance of over 3500 km (Section 4.2.1). By tracing the geographic path, we are able to automatically flag such anomalous routes, which would be difficult to do using purely network-centric information such as delay. We compute the *linearized distance* between two hosts as the sum of the geographic lengths of the individual links of the path. We then compute the ratio of the linearized distance of the path to the geographic distance between the source and destination hosts, which we term the *distance ratio*. A large ratio would be indicative of a circuitous and

acteristics using an overlay network. By actively monitoring the quality of different paths, their alternate path selection mechanism can quickly recover from network failures and optimize application specific performance metrics.

Consistent with these findings, our measurements indicate the existence of highly circuitous paths in the Internet. We also find that the circuitousness of a path is correlated with the minimum end-to-end latency along the path.

## 2.2 Topology discovery and mapping

Discovering and analyzing Internet structure has been the subject of many studies. Much of the work has focused on studying topology purely at the network level, without any regard to geography. Recently several tools have been developed to map network nodes to their corresponding geographic locations. A few Internet mapping projects have used such tools to incorporate some notion of geographic location in their maps.

The Mercator project [6] focuses on heuristics for Internet Map Discovery. The basic approach is to use traceroute-like TTL limited probe packets coupled with source routing to discover routers<sup>1</sup>. A key component of Mercator is the set of heuristics used to resolve *aliases*, i.e., multiple IP addresses corresponding to (possibly different interfaces on) a single router. The basic idea is to send a UDP packet to a non-existent port on a router and wait for the ICMP *port unreachable* response that it elicits. In general, the destination IP address of the UDP packet and the source IP address of the ICMP response may not match, indicating that the two addresses correspond to different interfaces on the same router. In our work we use geographic information to identify points of sharing in the network. We view this as complementary to network-level heuristics such as the ones employed in Mercator.

The Internet Mapping Project [2] at Bell Labs also uses a traceroute-based approach to map the Internet from a single source. The map is colored according to the octets of the IP address, so portions corresponding to the same ISP tend to be colored similarly. The map, however, is not laid out according to geography. Other efforts have produced topological maps that reflect the geography of the Internet. Examples include the MapNet [24] and Skitter [28] projects at CAIDA and the commercial Matrix.Net service [25].

A number of tools have been developed for determining the geographic location corresponding to an IP address. These tools use a variety of approaches to map an IP address to location: inferring location from *Whois*

records [7] (e.g., NetGeo [11]), extracting location information from traceroute data (e.g., GeoTrack [13], VisualRoute [30]), determining the location coordinates using delay measurements (e.g., GeoPing [13]), etc. Our previous work on IP2Geo [13] focused on developing several tools, including GeoTrack, to do IP-to-location mapping. In this work, we use the GeoTrack tool to analyze geographic properties of Internet routing.

## 3 Experimental methodology

In this section, we discuss our experimental methodology. We present the details of our measurement test bed and the data sets we gathered. We also discuss GeoTrack, the tool we used to determine geographic paths in the Internet.

### 3.1 Overview

Since the goal of our work is to study the geographic properties of Internet routing, much of our measurement work has focused on gathering network path data using the traceroute tool [8]. We are not interested in studying the dynamic properties of Internet routing (e.g., how routes change over time), so we only record a single snapshot of the network path between a given pair of hosts. It may be possible that some of the routes in our dataset are backup paths due to failures at the time of our measurement. However, we do not expect the aggregate statistics reported in this paper to be affected by such failures since our measurements were spread over a 2-month time period. We use traceroute to determine the network path between 20 traceroute sources and thousands of geographically distributed destination hosts.

Once we have gathered the traceroute data, we use the GeoTrack tool to determine the location of the nodes along each network path where possible. GeoTrack reports the location at the granularity of a city. We then use an on-line latitude-longitude server [18] to compute the geographic distance between the source and destination of a traceroute as well as between each pair of adjacent routers along the path. The latter enables us to compute the *linearized distance*, which we define as the sum of the geographic distances between successive pairs of routers along the path. So if the path between A and D passes through B and C, then the linearized distance of the path from A to D is the sum of the geographic distances between A & B, B & C, and C & D.

As we discuss in Section 3.4.1, we are typically able to determine the location of most but not all routers. We simply skip the routers whose locations we are unable to determine. So in the above example, if the location of C is unknown, then we compute the linearized distance of

<sup>1</sup>Actually, router *interfaces* are discovered, not routers.

possibly anomalous route. In Section 4, we study circuitousness of paths as a function of the geographic and network locations of the end-hosts.

Our results indicate that the presence of multiple ISPs in a path is an important contributor to circuitous routing. We also find intra-ISP routing to be far less circuitous than inter-ISP routing. Our study of circuitousness of paths provides some insights into the peering and routing policies of ISPs. Although circuitousness may not always relate to performance, it can often be indicative of a routing problem that deserves more careful examination.

There are two extremes to the routing policy that an ISP may employ: *hot-potato* routing and *cold-potato* routing. In hot-potato routing, the ISP hands off packets to the next ISP as quickly as possible. In cold-potato routing, the ISP carries packets on its own network as far as possible before handing them off to the next ISP. The former policy minimizes the burden on the ISP's network whereas the latter gives the ISP greater control over the end-to-end quality of service experienced by the packets. As we discuss in Section 5.4, geographic information provides a means to quantify these notions by using the geographic distance traversed within an ISP as a proxy for the amount of work performed by the ISP. In addition, we can also evaluate the degree to which an individual ISP contributes in the routing of packets end-to-end. Our analysis of properties of paths that traverse multiple ISPs is presented in Section 5.

Another aspect of routing that bears careful examination is its fault tolerance. Fault tolerance has generally been studied in the context of node or link failures based on network-level topology information. However, such topology information may be incomplete in that two seemingly independent nodes may actually be susceptible to correlated failures. For instance, a catastrophic event such as an earthquake or a major power outage might knock out all of an ISP's routers in a geographic region. Geographic information can help in identifying routers that are co-located. In order to analyze the impact of correlated failures, we consider ISP topologies at the geographic level, where each node represents a geographic region such as a city. Using the geographic topology information of several commercial ISPs gathered from CAIDA [24], we analyze the fault tolerance properties of individual topologies and the topology resulting from the combination of the individual ISP networks (Section 6). We find that many tier-1 ISPs are highly susceptible to single geographic node failures. The combined topology however exhibits better tolerance to such failures.

In summary, we believe geography is an interesting

means for analyzing and quantifying network properties. In some cases, our analysis provides additional evidence for existing intuition about certain properties of Internet routing (e.g., hot-potato routing, circuitous paths). An important contribution of our work is a methodology for quantifying such intuitions using geographic information. Such quantification enables us, for instance, to automatically flag circuitous paths, something that would be hard to using purely network-centric metrics (and no geographic information).

## 2 Related work

We classify related work into two categories: (a) Internet routing; (b) Topology discovery and mapping.

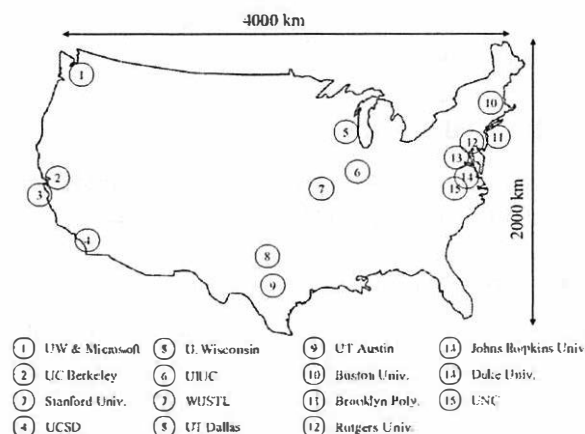
### 2.1 Internet routing

There are several properties of Internet routing that are of interest: end-to-end performance, routing stability, routing convergence, etc. Previous work on Internet routing has focused either on measuring these properties or on modifying certain aspects of routing with a view to improving performance. Our work shows how geographic information can be used to measure and quantify certain routing properties such as circuitous routing, hot-potato routing and geographic fault tolerance.

Network path information, obtained using the *traceroute* tool [8], has been used widely to study the dynamics of Internet routing. For instance, Paxson [14] studied various aspects of Internet routing using an extensive set of traceroute data. They include: routing pathologies, stability of routing, and routing asymmetry. In relation to our work, he studies circuitous routing by determining the geographic locations of the routers in his dataset and uses geographic distance as a metric to quantify it. In addition, he uses the number of different geographic locations along a path to analyze the effect of hot-potato routing as a potential cause for routing asymmetry. We extend this work by studying circuitousness as a function of the geographic and network location of end-hosts. We also analyze the effects of multiple ISPs in a path on its circuitousness. The distance ratio metric that we define can be used to automatically flag anomalies such as the large-scale route fluttering identified in [9, 14].

Overlay routing has been proposed as a means to circumvent the default IP routing. Savage et al. [17] study the effects of the routing protocol and its policies on the end-to-end performance as seen by the end-hosts. They show that for a large number of paths in the Internet, there exist paths that exhibit significantly better performance in terms of latency and packet loss rate. Recently, Andersen et al. [1] have proposed specific mechanisms for finding alternate paths with better performance char-

the path from A to D as the sum of the geographic distances between A & B and B & D. Clearly, skipping over C would lead us to underestimate the linearized distance. However, as noted in Section 3.4.1, most of the skipped nodes are in the vicinity of either the source or the destination, so the error introduced in the linearized distance computation is small.



**Figure 1: Locations of our traceroute sources in the U.S. Note that there were 17 hosts in 15 locations (two hosts each in Seattle and Berkeley).**

### 3.2 Measurement testbed

We used 20 geographically distributed hosts as the sources for our traceroutes. 17 of these hosts were located in the U.S. (Figure 1) while 3 were located in Europe (at Stockholm (Sweden), Bologna (Italy), and Budapest (Hungary)). The geographical diversity in source locations enables us to study the variations in routing properties as seen from different vantage points. For logistical reasons, it was convenient for us to locate the traceroute sources on university campuses. 18 out of the 20 traceroute sources fell into this category. Furthermore, 9 of the 15 university locations we considered in the U.S. were connected by the Internet2 backbone [19]. To add some diversity, we had one source in Berkeley, CA connected to a home cable modem network (in addition to a host at the University of California at Berkeley) and another in Seattle, WA connected to the Microsoft Research network (in addition to a host at the University of Washington at Seattle). These two pairs of sources allow us to study (albeit to a limited extent<sup>2</sup>) what impact, if any, the nature of the source's connectivity has.

The destination set for the traceroutes comprised several

<sup>2</sup>We could have used a diverse set of public traceroute servers [22] to overcome this limitation. However, the large volume of traceroutes that we were looking to run from each source precluded this.

thousand hosts. These destination hosts fell into 4 categories:

1. *UnivHosts*: 265 Web servers and other hosts located on university campuses in the U.S. The hosts were distributed across 44 of the 50 states in the U.S.
2. *LibWeb*: 1,205 Web servers of public libraries [21] distributed across 49 states in the U.S. We also ensured that the distribution of the geographic locations of these libraries is not skewed.
3. *TVHosts*: 3,100 client hosts in the U.S. that connected to an on-line TV program guide. A majority of these clients were located on non-academic networks such as America Online (AOL).
4. *EuroWeb*: 1,092 Web servers [23] distributed across 25 countries in Europe.

For ease of exposition, we sometimes refer to UnivHosts, LibWeb, and TVHosts as the U.S. hosts and EuroWeb as the European hosts.

This diverse set of destination hosts enables us to investigate the properties of Internet routing in the context of a large set of ISPs. In all, we traced approximately 84,000 end-to-end paths between our traceroute sources and the destination hosts during October-December 2000. Our data is available online at [27].

### 3.3 Dataset from 1995

To study the temporal variations in Internet properties, we use the traceroute data set collected by Paxson in 1995 [26]. The data set includes traceroutes conducted between pairs of hosts drawn from a set of 33 hosts distributed across (mainly academic sites in) the U.S., Europe, South Korea, and Australia.

Despite the fact that the 1995 data set contains far fewer paths than the 2000 data set, it provides an interesting data point for comparison. The 1995 data set was gathered in late 1995, about 6 months after the demise of the NSFNET backbone (which used to provide connectivity to academic sites in the U.S.) and early in the life of the commercial Internet.

### 3.4 GeoTrack

Once we have gathered traceroute data, we use the GeoTrack tool, which we developed previously as part of the IP2Geo project [13], to translate the network path between a pair of hosts to the corresponding geographic path. GeoTrack tries to infer the location of a router based on its DNS name. Network operators often assign



geographically meaningful names to routers<sup>3</sup>, presumably for administrative convenience. For example, the name *corerouter1.SanFrancisco.ca.net* corresponds to a router located in San Francisco. However, not all router names are *recognizable* (i.e., some router names may not contain an indication of location).

Here is a brief outline of how GeoTrack works; please refer to [13] for a more detailed description. The DNS name of the router is parsed to determine if it contains any location codes. GeoTrack uses a database of approximately 2000 location codes for cities in the U.S. and in Europe. Each ISP tends to use its own naming convention, so there may be multiple codes for each city (e.g., *chcg*, *chcgil*, *cgcil*, *chi*, *chicago*, *ord* for Chicago, IL). GeoTrack incorporates ISP-specific parsing rules that specify the subset of valid codes and the position(s) in which they may appear in the router names.

We use the domain name of a router to decide which ISP it belongs to. While this heuristic works reasonably well, it is not perfect because multiple domain names may correspond to the same administrative domain (e.g., *alter.net* and *uu.net*), often due to the merger of what were once independent networks. For the same reason, even AS numbers would not enable us to determine the administrative domain boundaries with complete accuracy.

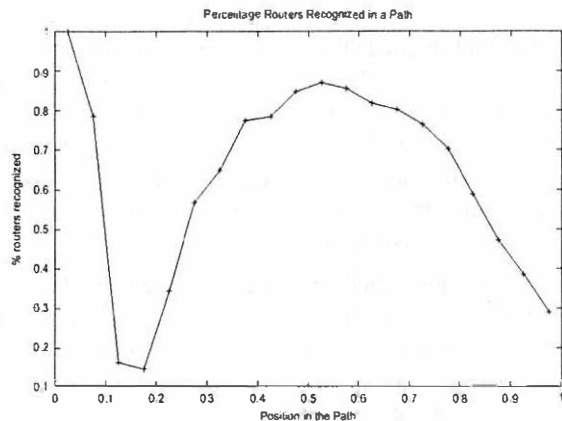
### 3.4.1 Coverage of GeoTrack

Of the 11,296 *.net* router names in our traceroute data set, 7842 were recognizable (approximately 70%). We compiled a list of 13 major ISPs with nationwide backbones in the U.S. or with international coverage: Sprintlink, AT&T, Cable and Wireless, Internet2, Verio, BBNPlanet<sup>4</sup>, Qwest, Level3, Exodus, PSINet, UUNET/Alter.net, VBNS, and Global Crossing. We found that 5,966 of the 6,859 router names for these major ISPs were recognizable (87%). In some individual cases, such as AT&T and UUNET, the recognizability was in excess of 95%.

By manual inspection, we found that a large chunk of the router names which are unrecognizable by our tool have no meaningful codes to decipher their locations. Many unrecognizable router names tend to be concentrated in regional or campus networks. (For example, *cmu.psc.net* is a node in Pittsburgh, PA. However, since it does not contain a valid city or airport code, GeoTrack is unable

<sup>3</sup>To be precise, DNS names are associated with router *interfaces*, not routers themselves. However, for ease of exposition we simply use the term router.

<sup>4</sup>BBNPlanet is now called Genuity, but the router names are still in the *bbnplanet.net* domain.



**Figure 2: The recognizability of router names as a function of the position of the router in the end-to-end path. The position is quantified by dividing the number of hops leading up to the router by the total number of hops end-to-end.**

to recognize its location.<sup>5</sup>) Figure 2 shows that recognizability is lowest close to the start and the end of the path. (The peak corresponding to the very beginning of the path is due to the source location always being known.) Thus most of the unrecognizable nodes are typically located in the vicinity of the source or the destination, so the resulting error in linearized distance is minimal.

In the case of the 1995 data set, GeoTrack is able to recognize 1,289 out of 1,531 router names (approximately 84%). Interestingly, we noticed a huge difference in the naming convention used in 1995 and 2000. Hence we needed to create a new set of codes for the 1995 data set.

### 3.4.2 Possible inaccuracies

First, the city codes used in GeoTrack for computing the location of router given its label are manually determined and encoded. Hence there is always a possibility that the location of a router as determined by GeoTrack is incorrect. However, we have greatly reduced the possibility of such errors by using delay-based verification, ISP specific parsing rules and manual inspection. In delay-based verification, we perform the following simple check: if the difference between the minimum RTTs to two adjacent routers in a path is not high, the distance between them cannot be large. This simple check helped us distinguish between two cities named *Geneva* that had similar city codes — one in Switzerland and the other in

<sup>5</sup>Of course, it is possible to include *psc* and *cmu* as codes. However, we refrain from doing so since we only want to include those codes in GeoTrack that inherently indicate location. Doing otherwise would lead us down the path of exhaustive tabulation, which is undesirable.

Texas. We have enumerated specific rules for 52 different ISPs (all major ISPs in our data set) which specify the exact position where a city code is embedded in a label. This, in conjunction with ISP specific city-codes, greatly reduces the chances of a wrong location output. We have also manually inspected the geographic paths corresponding to a large sample of our traceroute data to check for any possible errors.

Second, the linearized distance computed can be distorted if the geographic locations of many routers in a path are unknown. We reduce this distortion by restricting our analysis to paths that have at least 4 recognizable intermediate routers. The linearized distance of a path can also be skewed due to intra-metro distances. Intra-metro distances will affect our analysis only for small values of linearized distances. To reduce this skew, we only consider paths with a linearized distance greater than 100 kms in our study.

### 3.5 Limitations

We now discuss the limitations of our study arising both due to the inherent limitations of geographic information and due to limitations of our experimental methodology.

#### 1. Geography does not determine performance:

There is not a perfect relationship between geographic distance and network performance. It is possible that a circuitous path yields better performance than a less circuitous one. For instance, the most optimal path between certain countries may be via the U.S. even if that means a large detour in geographic terms. However, in Section 4.5, we show that there exists a strong correlation between the minimum end-to-end delay between two end-hosts and the linearized distance of their connecting path. In light of this, we view our geographic analysis of network paths as providing (a) hints on paths that are *potentially* anomalous and should be examined more closely to determine if they are indeed anomalous, (b) an indication of how much improvement there could be in end-to-end latency if a non-circuitous path between source and destination were feasible, and (c) a way to quantify network properties such as hot-potato routing, which may provide new insight into these properties.

#### 2. IP-level topology is incomplete:

Our linearized distance computation only considers the router-level (i.e., IP-level) topology. We have no way of discovering the underlying physical topology (which may be based on ATM, SONET, or other technologies), so in general we would underestimate the linearized distance. While this is a limitation of our methodology, we note that the

trend in high-speed networks (OC-48 and faster) is away from separate layer-2 and layer-3 architectures (e.g., IP-over-ATM) and towards an all-IP network [15]. This trend increases the applicability of our methodology.

## 4 Circuitousness of Internet paths

In this section, we examine the nature of circuitous routes in the Internet. Since there is not a standard measure of circuitousness, we define a metric, *distance ratio*, as the ratio of the linearized distance of a path to the geographic distance between the source and destination of the path. The distance ratio reflects the degree to which the network path between two nodes deviates from the direct geographic path between the nodes. A ratio of 1 would indicate a perfect match (i.e., an absolutely direct route) while a large ratio would indicate a circuitous path.

We present several different analyses with a view to studying the impact of spatial factors as well as temporal factors. Under spatial factors, we study the effect of the geographic and network locations of end-hosts on the circuitousness of paths. To study temporal properties, we compare the circuitousness of paths drawn from Paxson's 1995 data set to the ones drawn from our 2000 data set. Finally, we analyze the relationship between the minimum delay between two end-hosts and the linearized distance along their path.

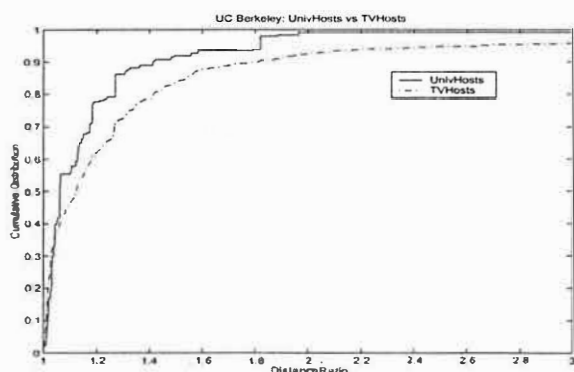
### 4.1 Effect of network location

In this section, we will vary the network location of the end-hosts (source and destination) and study its effect on the distance ratio of paths. In our first analysis, we fix a source and compare the distance ratio of paths to destinations in different networks. In our second analysis, we compare the distance ratio of paths from different sources in the same geographic location but with different network connectivities to a set of end-hosts in the same network.

#### 4.1.1 Paths from a single source

We consider paths from our traceroute sources in U.S. universities to two varied sets of end-hosts: UnivHosts and TVHosts. Many of the hosts in UnivHosts (including our sources) connect to the Internet2 high-speed backbone via a local GigaPOP. So much of the wide-area path between our sources and a host in UnivHosts traverses the Internet2 backbone. On the other hand, TVHosts is a more diverse set that includes hosts located in various commercial networks (AOL, MSN, @Home, etc.) as well as university campuses. So the wide-area paths

from our sources to the hosts in TVHosts typically traverse one or more commercial ISP backbones.



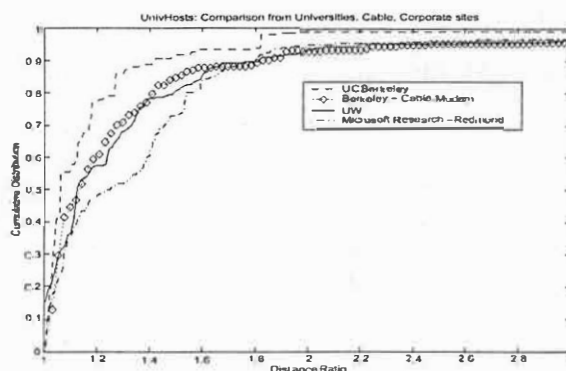
**Figure 3: CDF of distance ratio for paths from UC Berkeley to UnivHosts and TVHosts.**

This difference between the two groups of destination hosts is reflected in the cumulative distribution function (CDF) of the distance ratio for the two cases. As Figure 3 shows (for source in UC Berkeley), the distance ratio is close to 1 for many of the destinations. The ratio is 1.1 or less (corresponding to a linearized distance that exceeds the end-to-end geographic distance by no more than 10%) for 55% of the destinations in UnivHosts and 45% in TVHosts. This finding is consistent with the rich Internet connectivity of the San Francisco Bay Area (where UC Berkeley is located). The area includes several public Internet exchanges (e.g., MAE-West, PAIX, etc.) as well as private peering points. So a path from the UC Berkeley host to a destination host is often (but not always) able to transition to the latter's ISP within the SF bay area itself. So there is little need to take a detour through another city just to transition to the destination's ISP.

There is a far more pronounced difference between the UnivHosts and TVHosts cases if we look at the tail of the distribution. For instance, at the 90th percentile mark, the distance ratio is 1.41 in the case of UnivHosts but 1.72 in the case of TVHosts; in other words, the detour is 1.75 times as large for TVHosts destinations as it is for UnivHosts (72% versus 41%). The paths to some of the hosts in TVHosts tend to be more circuitous because they traverse multiple commercial ISPs whose peering relationships may cause detours in the end-to-end path. We discuss this issue in more detail in Section 5. We observe qualitatively the same trends for other university sources as well; i.e., the distance ratio tends to be smaller for paths leading to UnivHosts compared to TVHosts.

#### 4.1.2 Multiple sources in the same location

We now consider paths from pairs of hosts in the same location but on entirely different networks to destinations in the UnivHosts set. We consider two such pairs of traceroute sources: (a) a machine on the Berkeley campus and another also in Berkeley but on @Home's cable modem network, and (b) a machine at the University of Washington (UW) campus in Seattle and another on the Microsoft Research network 10 km away.



**Figure 4: CDF of distance ratio for paths from pairs of co-located sources to UnivHosts.**

Figure 4 shows the CDF of the distance ratio for all 4 sources. For the two sources located in Berkeley, we find that the one on the university campus has a significantly smaller distance ratio, especially at the tail of the distribution. For instance, the 90th percentile of the distance ratio for the UC Berkeley source is 1.41 while that for the cable modem source is 1.83. Since the destination set is UnivHosts, the UC Berkeley source tends to have more direct routes (via Internet2) than the cable modem client has (via @Home and other commercial ISPs).

We observe a similar trend for the UW-Microsoft pair. The UW source has more direct routes to other university hosts than does the Microsoft source. For instance, the path from Microsoft to the University of Chicago follows a highly circuitous route through BBNPlanet's (Genuity) network. The geographic path traversed includes Los Angeles, Carlton (TX), Indianapolis and Chicago (in that order). The linearized distance of the path is 4976 km while the geographic distance between Seattle and Chicago is only 2795 km. In contrast, the path from UW (via Internet2) is far more direct: it passes through Denver, Kansas City, Indianapolis, and finally Chicago, for a total linearized distance of 3533 km.

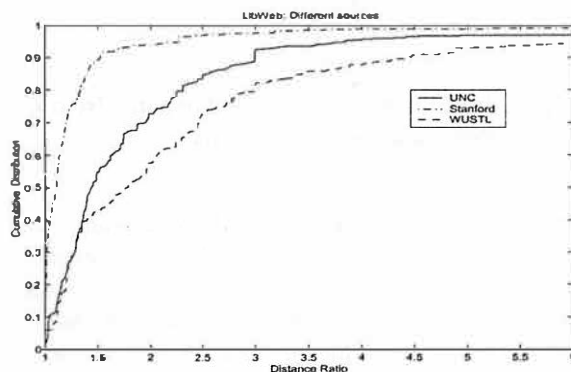
These results indicate that the nature of network connectivity of the source and the destination has a significant impact on how direct or circuitous the network paths are.

## 4.2 Effect of geographic location

The geographic location of a source indirectly determines its network connectivity. Sources near well-connected geographic locations like the Bay Area can potentially have less circuitous routes since many commercial ISPs will have a POP very close to them. To better understand the effect of geographic location, we compare the distance ratios of sources in different locations to a common set of destination end-hosts. We extend this analysis to study the role of network structures in different continents (U.S. and Europe) on the circuitousness of paths.

### 4.2.1 Multiple sources in different locations

We consider paths from sources in three geographically distributed locations in the U.S.: Stanford, Washington University at St. Louis (WUSTL), and the University of North Carolina (UNC). The destination set is LibWeb, which is a larger and more diverse set than the UnivHosts set considered in Section 4.1.2.



**Figure 5: CDF of distance ratio for paths from multiple sources to LibWeb.**

As shown in Figure 5, the distance ratio tends to be the smallest for paths originating from Stanford and the largest for those originating from WUSTL. Stanford, like Berkeley, is located in the San Francisco Bay area, which is well served by many of the large ISPs with nationwide backbones. In contrast, WUSTL is much less well connected. Almost all paths from WUSTL enter Verio's network in St. Louis and then take a detour either to Chicago in the north or Dallas in the south. At one of these cities, the path transitions to another major ISP such as AT&T, Cable & Wireless, etc. and proceeds to the destination. Any detour is particularly expensive in terms of the distance ratio because the central location of St. Louis in the U.S. means that the geographic distance to various destinations is relatively small.

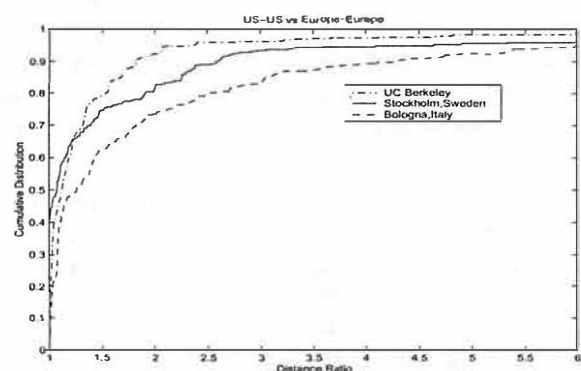
In general, paths (such as those from WUSTL) that traverse significant distances in the backbones of two or

more large ISPs tend to be more circuitous than paths (such as those from Stanford) that traverse much of the end-to-end distance in the backbone of a single ISP (regardless of who the ISP is). One example of a highly circuitous path we found involved two large ISPs, Verio and AT&T. The path originates in WUSTL in St. Louis and terminates at a host in Indiana University, 328 km away. However, the geographic path goes from St. Louis to New York via Chicago, all on Verio's network. In New York, it transitions to AT&T's network and then retraces its path back through Chicago to St. Louis, before finally heading to Indiana. The linearized distance is 3500 km, more than 10 times as much as the geographic distance. We examine the impact of multiple ISPs in greater detail in Section 5.

While the specific findings pertaining to Stanford and WUSTL may not be important in general, our results suggest that the distribution of the distance ratio is consistent with our intuition about the richness of connectivity of hosts in different geographic locations.

### 4.2.2 U.S. versus Europe

We now analyze the distance ratios for paths in Europe and compare these to the distance ratios for paths in the U.S. We consider paths from the 17 U.S. sources to destinations in the LibWeb set and also paths from the 3 European sources to destinations in the EuroWeb set. Thus, all of these paths are contained either entirely within the U.S. or entirely within Europe. We do not consider paths from U.S. sources to European destinations (or vice versa) because the distance ratio for such paths tends to be dominated by long transatlantic links (which tends to push the ratio towards 1).



**Figure 6: CDF of distance ratio for paths within the U.S. and those within Europe.**

In Figure 6, we show the distribution of the distance ratio for three sources: Berkeley in the U.S., and Stockholm (Sweden) and Bologna (Italy) in Europe. We observe that the distance ratio tends to be larger for the Eu-

ropean sources compared to Berkeley, especially in the tail of the distribution. We attribute this to three causes.

First, paths in Europe tend to traverse multiple regional or national ISPs. The complex peering relationships between these ISPs often results in convoluted paths. For instance, a path from Bologna to a host in Salzburg, Austria traverses 3 ISPs – GARR (Italian Academic and Research Network), Equip/Infonet, and KPNQwest (a leading pan-European ISP based in the Netherlands) – and passes through Milan (Italy), Geneva (Switzerland), Paris (France), Amsterdam (Netherlands), Frankfurt (Germany), and Vienna (Austria). The linearized distance of the path is 2506 km whereas the geographic distance between Bologna and Salzburg is only 383 km.

Second, in some cases the path from a European source to a European destination passes through nodes in the U.S. For instance, a path from Stockholm (Sweden) to Zagreb (Croatia) passes through a node in New York City belonging to Teleglobe, a large international ISP. In the event that the ISPs in Europe have better connectivity to ISPs in U.S., it would be appropriate for them to route their traffic through U.S. though the route may be more circuitous. Third, geographic distances in Europe tend to be smaller than the ones in U.S. As in the case of St Louis in Section 4.2.1, small detours in routing can be particularly expensive in terms of the distance ratio for paths between end-hosts in Europe.

### 4.3 Temporal properties of routing

To better understand some of the temporal properties of routing, we compare the distribution of the distance ratio computed from our 2000 data set with that computed from Paxson's 1995 data set [20]. The paths in the 1995 data set correspond to traceroutes conducted amongst the 33 nodes (mainly at academic locations) that were part of the testbed. We considered 340 paths between the subset of 20 nodes that were located in the U.S. The 1995 data set includes multiple traceroute measurements between each pair of hosts. In our study, we only use data from one successful traceroute between each pair of hosts. To keep the nature of the measurement points similar, in the 2000 data set we only consider paths between the 15 source hosts located at universities and the 265 hosts in the UnivHosts set.

Figure 7 plots the CDF of the distance ratio for the 1995 and 2000 data sets. By observing the tail of the cumulative distribution, we find that the distance ratios tend to be smaller in the 2000 data set. This improvement is not surprising because the Internet is more richly connected today than it was 5 years ago. There now exist direct point-to-point links between locations that were previously connected only by an indirect path.

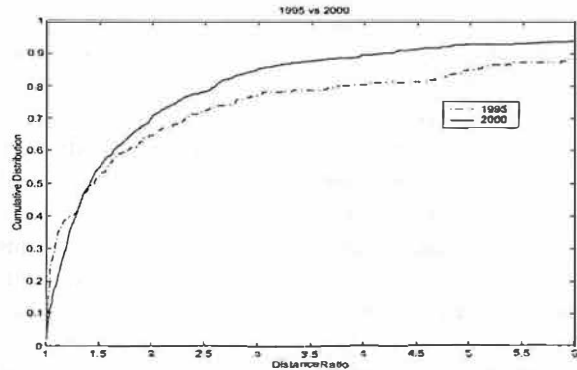


Figure 7: CDF of distance ratio for paths in Paxson's 1995 data set and our data set from 2000.

### 4.4 Correlation between delay and distance

Finally, we analyze the relationship between geography and the end-to-end delay along a path. Though geography by itself cannot provide any information about many performance characteristics like bandwidth, congestion along a path, the linearized distance of a path does enforce a minimum delay along a path (propagation delay along a path).

To study this correlation, we use the TVHosts data set since it represents a wide variety of end-hosts. In our traceroute data, we obtain 3 RTT samples for every router along the path. Since not all routers in a path are recognizable, we consider the minimum RTT, geographic distance and linearized distance to the last recognizable router along the path. In this analysis, we restrict ourselves to the list of probes in the U.S.

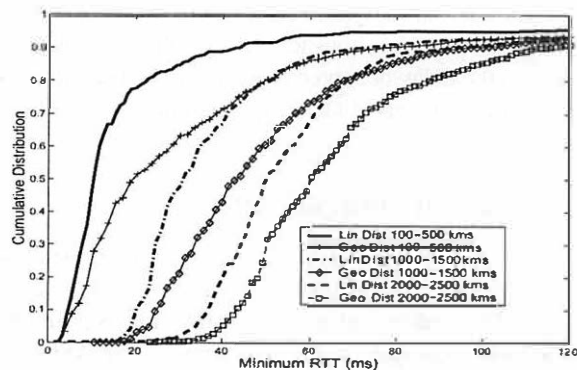


Figure 8: CDF of minimum end-to-end RTT to TVHosts for different ranges of linearized distances and geographic distances of paths

Figure 8 illustrates the correlation of the minimum RTT along a path to the linearized distance of a path and the geographic distance between the end-hosts. We make



three important observations. First, at low values of the linearized distance there exists a strong correlation between the delay and linearized distance for a large fraction of end-hosts especially for small values of linearized distances. We expect this correlation to be much stronger as we compute the minimum over a larger number of samples. Second, linearized distance along a path does enforce a minimum end-to-end RTT which is an important performance metric for latency sensitive applications. Third, the minimum RTT between two end-hosts has lesser correlation to the geographic distance between them as compared to the linearized distance of the path connecting them. We observe that for a given range of linearized distance of a path, the RTT variation is much smaller than its variation for the same range of geographic distance between the end-hosts. Hence linearized distance of a path conveys more about the minimum RTT characteristics of a path than merely the geographic distance between the end-hosts. We also verified that these observations hold across the other data sets we collected. The coarse correlation between minimum delay and geographic distance was used in building GeoPing, an IP-to-location mapping service [13].

#### 4.5 Summary of Results

From Sections 4.1 and 4.2, we observe that the circuitousness of a route depends on both the geographic and network location of the end-hosts. In many cases, the trends we observe in the distance ratio are consistent with our intuition. A large value of the distance ratio enables us to automatically flag paths that are highly circuitous, possibly (though not necessarily) because of routing anomalies. Finally, we show that the minimum delay between end-hosts and the linearized distance of their path are strongly correlated. This relationship indicates that the circuitousness of a route does have an effect on the delay observed along the route (though this does not completely dictate the performance along the route).

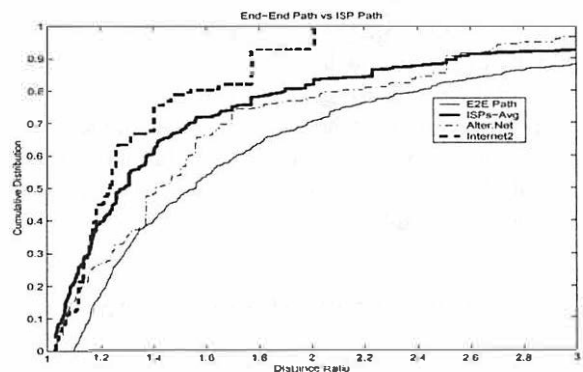
### 5 Impact of multiple ISPs

Our analysis in Section 4 focused on the characteristics of the end-to-end path from a source to a destination. The end-to-end path typically traverses multiple autonomous systems (ASes). Some of the ASes are stub networks such as university or corporate networks (where the source and destination nodes may be located) whereas others are ISP networks. The relationships between these networks is often complex. There are customer-provider relationships (such as those between a university network and its ISP or between a regional ISP and a nationwide ISP) and peering relationships (such as those between two nationwide ISPs). A

stub network may be multi-homed (i.e., be connected to multiple providers). Two nationwide ISPs may peer with each other at multiple locations (e.g., San Francisco and New York).

These complex interconnections between the individual networks have an impact on end-to-end routing. In this section, we show that geography can indeed be used as a means to analyze these complex interconnections. Specifically, we investigate the following questions: (a) are Internet paths within individual ISP networks as circuitous as end-to-end paths?, (b) what impact does the presence of multiple ISPs have on the circuitousness of the end-to-end path?, (c) what is the distribution of the path length within individual ISP networks, and (d) can geography shed light on the issue of hot-potato versus cold-potato routing?

#### 5.1 Circuitousness of end-to-end paths versus intra-ISP paths



**Figure 9: CDF of distance ratio of end-to-end paths versus that of sections of the path that lie within individual ISP networks.**

We now take a closer look at the circuitousness of end-to-end Internet paths, as quantified by the distance ratio. We compare the distance ratio of end-to-end paths with that of sections of the path that lie within individual ISP networks. We consider paths from the U.S. sources to the LibWeb data set for this analysis.

As shown in Figure 9, the distance ratio of end-to-end paths tend to be significantly larger than that of intra-ISP paths. In other words, end-to-end paths tend to be more circuitous than intra-ISP paths. Furthermore, the distribution of the ratio tends to vary from one ISP to another, with Internet2 doing much better than the average and Alter.Net (part of UUNET) doing worse.

We believe the reason that end-to-end paths tend to more circuitous is that the peering relationship between ISPs may create detours that would otherwise not be present. Inter-domain routing in the Internet largely uses the

BGP [16] protocol. BGP is a path vector protocol that operates at the level of ASes. It offers limited visibility into the internal structure of an AS (such as an ISP network). So the actual cost of an AS-hop (in terms of latency, distance, etc.) is largely hidden at the BGP level. As a result the end-to-end path may include large detours.

Another issue is that ISPs typically employ BGP policies to control how they exchange traffic with other ISPs (i.e., which traffic enters or leaves their network and at which ingress/egress points). The control knobs made available by BGP include import policies such as assigning a local preference to indicate how favorable a path is and export policies such as assigning a multiple exit discriminator to control how traffic enters the ISP network [5]. These policies are often influenced by business considerations. For instance, packets from a customer of ISP A to a customer of ISP B in the same city might have to go via a peering point in a different city simply because a local service provider in the origin city who peers with both ISP A and ISP B does not provide transit service between the two ISPs.

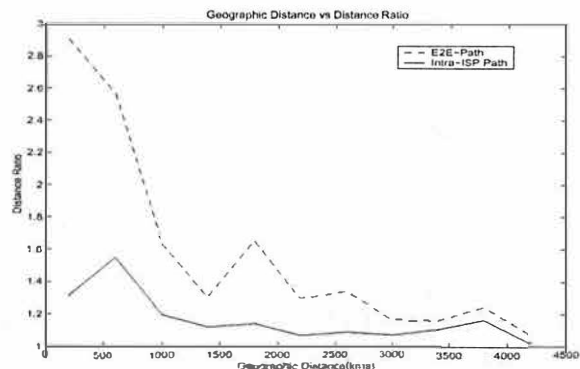
Such BGP policies may partly explain the example mentioned in Section 4.2.1, where packets from a host in St. Louis to a nearby location had to travel on Verio's network all the way to New York to enter AT&T's network. We have seen several other such examples: a path from Austin, TX to Memphis, TN where the transition from Qwest to Sprintlink happens in San Jose, CA; a path from Madison, WI to St. Louis, MO where the transition from BBNPlanet to Qwest happens in Washington DC. We do not have specific information on the policies that were employed by these ISPs, so we cannot make a definitive claim that BGP is to blame. However, in view of the complex policies that come into play in the context of inter-domain routing, it is not surprising that end-to-end paths tend to be more circuitous.

In contrast, routing within an ISP network is much more controlled. Typically, a link-state routing protocol, such as OSPF [12], is used for intra-domain routing. Since the internal topology of the ISP network is usually known to all of its routers, routing within the ISP network tends to be close to optimal. So the section of an end-to-end path that lies within the ISP's network tends to be less circuitous. Referring again to the example in Section 4.2.1, both the St. Louis → Chicago → New York path within Verio's network and the New York → Chicago → St. Louis path within AT&T's network are much less circuitous than the end-to-end path.

However, this does not mean that intra-ISP paths are never circuitous. As noted in Section 4.1.2, we found a circuitous path through BBNPlanet (Genuity), from Mi-

crosoft Research in Seattle to the University of Chicago, that has a linearized distance of 4976 km whereas the geographic distance is only 2795 km. This does not imply that the path is necessary sub-optimal. In fact, the circuitous path may be best from the viewpoint of network load and congestion. The point is that while geography provides useful insights into the (non-)optimality of network paths, it only presents part of the picture.

### 5.1.1 Impact of path length on circuitousness



**Figure 10: Distance ratio versus the geographic distance between the ends of a path. The median distance ratio is computed over 400 km buckets (0-400 km, 400-800 km, and so on). A minimum distance threshold of 100 km is imposed to prevent the ratio from blowing up, so the first bucket is actually 100-400 km.**

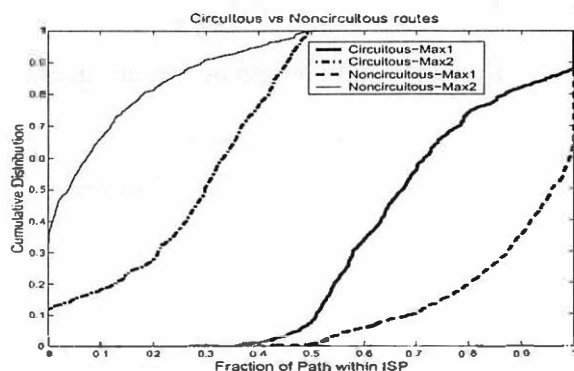
One question that arises from the above analysis is whether there is a connection between the circuitousness of a path and its length (i.e., the geographic distance between the two ends of the path). In other words, are longer paths inherently more circuitous, regardless of whether they traverse one ISP or many? If so, the fact that end-to-end paths tend to be longer than intra-ISP paths may explain the greater circuitousness of the former.

However, as shown in Figure 10, the trend is quite the opposite. The distance ratio tends to decrease as the geographic distance increases.<sup>6</sup> The reason is that the impact of a detour is smaller (in relative terms) in the context of a longer path. The distance ratio for the end-to-end path tends to be greater than that for the intra-ISP path,

<sup>6</sup>The jaggedness of the curves arises because of the large variance in distance ratio for small values of geographic distance. The 5th and 95th percentile marks for the 100-400 km bucket are (1.00, 20.50) for the end-to-end case and (1.00, 4.22) for the intra-ISP case. The corresponding marks for the 4000-4400 km bucket are (1.01, 1.57) for the end-to-end case and (1.00, 1.18) for the intra-ISP case.

regardless of geographic distance. Thus the greater circuitousness of end-to-end paths is most likely due to the presence of multiple ISP networks in the path.

## 5.2 Impact of multiple ISPs on circuitousness

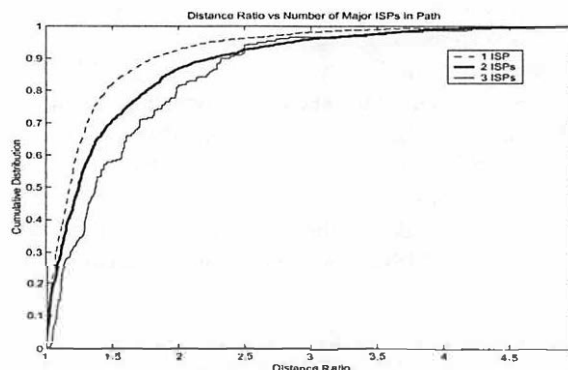


**Figure 11: CDF of the fraction of the end-to-end path that lies within the top 2 ISPs in the case of circuitous paths and non-circuitous paths.**

In Section 5.1 we hypothesized that the presence of multiple ISPs in an end-to-end path contributes to the circuitousness of the path. We now examine this issue more carefully. We classify end-to-end paths into two categories – non-circuitous (distance ratio  $< 1.5$ ) and circuitous (distance ratio  $> 2$ ).<sup>7</sup> For each path in either category, we identify the top two ISPs that account for most of the end-to-end linearized distance. We then compute the fraction of the end-to-end linearized distance that is accounted for by the top two ISPs, and denote these fractions by  $\max_1$  and  $\max_2$ . For example, if an end-to-end path with a linearized distance of 1000 km traverses 400 km in AT&T's network and 300 km in UUNET's network (and smaller distances in other networks), then  $\max_1 = 0.4$  and  $\max_2 = 0.3$ . Note that it is possible for  $\max_1$  to be 1.0 (and so  $\max_2$  to be 0.0) if the entire end-to-end path traverses just one ISP network. We note that local-area networks confined to a city (e.g., a university network) contribute nil to the linearized distance and therefore are ignored.

Figure 11 shows the CDF of  $\max_1$  and  $\max_2$  for the circuitous and non-circuitous paths. The difference in the characteristics of these two categories of paths is striking. The  $\max_1$  and  $\max_2$  curves are much closer together in the case of circuitous paths than in the case of

non-circuitous paths. In other words, in the case of circuitous paths, the end-to-end path traverses substantial distances in each of the top two ISPs (and perhaps other ISPs too). In contrast, non-circuitous paths tend to be dominated by a single ISP. For instance, the median values of  $\max_1$  and  $\max_2$  in the case of circuitous paths is approximately 0.65 and 0.3, respectively. In other words, the top two ISPs account for 65% and 30%, respectively, of the end-to-end path in the median case. However, the fractions for the non-circuitous paths are approximately 95% and 4%, respectively – much more skewed in favor of the top ISP.



**Figure 12: CDF of the distance ratio as a function of the number of major ISPs traversed along an end-to-end path. There were few paths that traversed more than 3 major ISPs.**

We also consider the impact of the number of *major* ISPs traversed along an end-to-end path on the distance ratio. Figure 12 shows a clear trend: the distance ratio tends to increase as the path traverses a greater number of ISPs. For instance, the median distance ratios are 1.18, 1.25, and 1.38, respectively with 1, 2, and 3 major ISPs. The 90th percentile of the distance ratio is 1.81, 2.26, and 2.35, respectively. A path that traverses a larger number of major ISPs may span a greater distance. However, as noted in Section 5.1.1, this would not explain the larger distance ratio. In fact, a greater geographic distance would tend to make the distance ratio smaller, not larger.

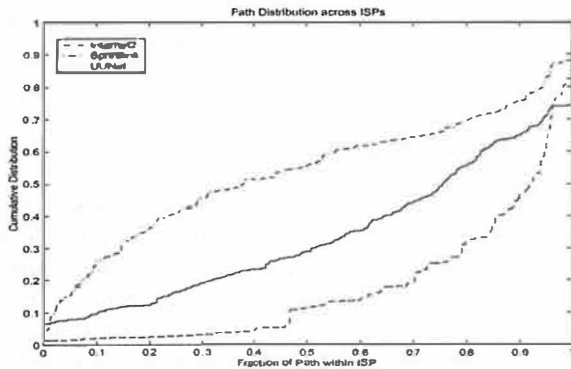
These findings reinforce our hypothesis that there is a correlation between the circuitousness of a path (as quantified by the distance ratio) and the presence or absence of multiple ISPs that account for substantial portions of the path.

## 5.3 Distribution of ISP path lengths

In this section, we further examine the distribution of the end-to-end linearized distance that is accounted for by individual ISPs. We wish to understand how the effort

<sup>7</sup>While the choice of these thresholds is arbitrary, they capture the intuitive notion of circuitous and non-circuitous routes. Note that there may be paths that do not fall into either category.

of carrying traffic end-to-end over a wide-area path is apportioned between different ISPs. For each of the 13 nationwide ISPs in the U.S. listed in Section 3.4.1, we consider the set of paths that traverse one or more nodes in that ISP's network. For each such path, we compute the fraction of the end-to-end path that lies within the ISP's network.



**Figure 13: CDF of the fraction of the end-to-end path that lies within individual ISP networks.**

Figure 13 plots the CDF of this fraction for a few ISPs. In each case, we consider the paths from the U.S. university sources to the LibWeb data set. We observe that the distributions look very different. For instance, the median fraction of the end-to-end path that lies within Sprintlink is only about 0.35 whereas the corresponding fraction for UUNET is 0.75 and for Internet2 is over 0.9. Internet2 is a high-speed backbone network that connects many university campuses in the U.S. An end-to-end path that traverses Internet2 typically originates and terminates at university campuses. Therefore, the Internet2 backbone accounts for an overwhelming fraction of such end-to-end paths. UUNET accounts for a larger fraction of the paths that traverse its backbone than any other commercial ISP we considered. This may reflect the close relationship between UUNET's parent company, Worldcom (which runs the vBNS backbone [29]), and academic sites.

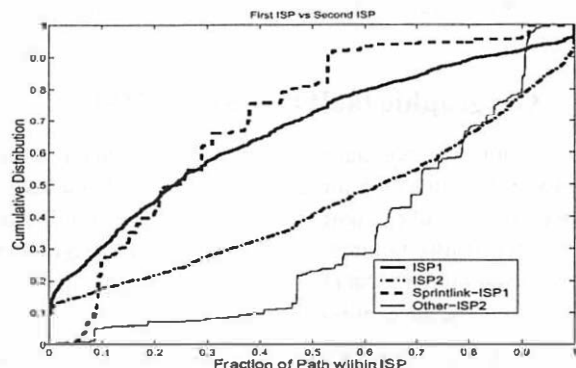
The much smaller fraction in the case of Sprintlink is harder to explain definitively. From our conversations with people at Sprint [3, 10], we have learned that academic sites are not their major customers, so Sprintlink participates minimally in carrying academic traffic. The location of our traceroute sources at academic sites may explain why Sprintlink only accounts for a small fraction of the end-to-end path.

We stress, however, that the point of our analysis is not to make general claims about certain ISPs being better or worse than others. Rather it is to show that geographic analysis of end-to-end paths yields interesting insights

into the role played by multiple ISPs in specific contexts (e.g., academic sites) and that these insights are consistent with our intuition.

## 5.4 Hot-potato versus Cold-potato routing

Finally, we investigate whether geographic information can be helpful in assessing whether ISP routing policies in the Internet conform to either hot-potato routing or cold-potato routing. In hot-potato routing, an ISP hands off traffic to a downstream ISP as quickly as it can. Cold-potato routing is the opposite of hot-potato routing where an ISP carries traffic as far as possible on its own network before handing it off to a downstream ISP. These two policies reflect different priorities for the ISP. In the hot-potato case, the goal is to get rid of traffic as soon as possible so as to minimize the amount of work that the ISP's network needs to do. In the cold-potato case, the goal is carry traffic on the ISP's network to the extent possible so as to maximize the control that the ISP has on the end-to-end quality of service. In general, an ISP's routing policy would lie somewhere in between the extremes of hot-potato and cold-potato routing.



**Figure 14: CDF of the fraction of the end-to-end path that lies within the first and second ISP networks in sequence.**

We consider the set of paths from U.S. sources to TVHosts. For each path that traverses two or more major ISPs (with nationwide backbones), we compute the fraction of the end-to-end path that lies within the first major ISP (ISP1) and the second major ISP (ISP2) in sequence. We use these fractions as measures of the amount of work that these ISPs do in conveying packets end-to-end. The distributions of these fractions is plotted in Figure 14. We observe that the fraction of the path that lies within the first ISP tends to be significantly smaller than that within the second ISP. For instance, the median is 0.22 for the first ISP and 0.64 for the second ISP. This is consistent with hot-potato routing behavior because the first ISP tends to hand off traffic quickly to the second ISP who carries it for a much greater distance.

Figure 14 also plots the distributions of the path lengths in the case where the first ISP is Sprintlink. We find that the difference between the ISP1 and ISP2 curves is even greater in this case. Again, this is consistent with hot-potato routing behavior on the part of Sprintlink for routes from academic locations.

## 5.5 Summary

In this section, we have used geographic information to study various aspects of wide-area Internet paths that traverse multiple ISPs. We found that end-to-end Internet paths tend to be more circuitous than intra-ISP paths, presumably because of the peering relationships between ISPs. Furthermore, paths that traverse substantial distances within two or more ISPs tend to be more circuitous than paths that largely traverse only a single ISP. Some of this circuitous routing behavior can be attributed to sub-optimal geographic peering between ISPs. Finally, the findings of our geography-based analysis are consistent with the hypothesis that ISPs generally employ hot-potato routing. The presence of hot-potato routing may also explain for why some major ISPs only account for a relatively small fraction of the end-to-end path.

## 6 Geographic fault tolerance of ISPs

An important component of studying Internet routing is to understand its fault tolerance aspects. Fault tolerance of a network is normally studied at the granularity of router or link failures. However such a failure model does not capture the fact that two seemingly independent routers can be susceptible to correlated failures.

We ask the question: what is the tolerance of an ISP's network to a *total* network failure in a geographic region, i.e., a failure that affects all paths traversing the region? We refer to such a failure as a *geographic failure*. Potential reasons for such a failure include natural calamities such as earthquakes or power blackouts.

By using the geographic location information of the routers, we can identify routers that are co-located and thereby construct a *geographic topology* of an ISP. In this topology, each geographic region is associated with a node and an edge between two nodes signifies the existence of at least one long-haul backbone link that connects the corresponding geographic regions.

We obtained the geographic topologies for 9 of the 13 major ISPs listed in Section 3.4.1 from the CAIDA MapNet site [24]. These are: AT&T, Cable and Wireless, Sprintlink, Genuity, Qwest, PSINet, UUNet, Verio and Exodus. Many of these topologies are obtained from information published at the ISPs' Web sites and are between 6-12 months out of date. Although it may be

possible to construct an ISP's geographic topology using extensive traceroute measurements, it would be hard to assess the completeness of the constructed topology. Hence we restrict ourselves to the geographic topologies obtained from CAIDA. However, as acknowledged by CAIDA [24], it is possible that these topologies may themselves be incomplete. This may be due to limited tracing or the presence of backup paths in routing. We will perform our analysis under the assumption that these topologies are reasonably complete and only have a few missing links.

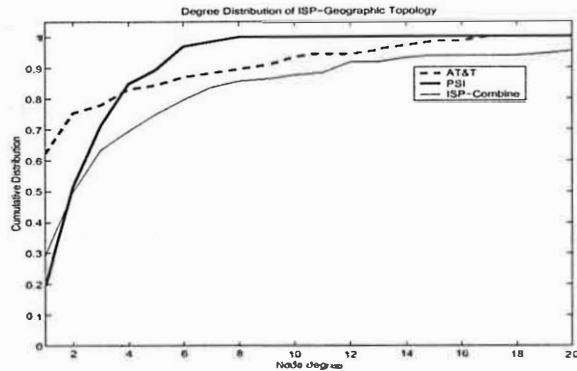
### 6.1 Degree distributions

The degree of a node provides a first-level quantification of the fault tolerance of that node in a given topology. A node with a degree  $k$  can tolerate up to  $k$  geographic failures before getting completely disconnected from all other nodes in the topology. In particular, a leaf node is not resilient to the geographic failure of its neighbor, but the failure of a leaf node itself has minimal impact on the rest of the network. On the other hand, the failure of a node with a very high degree would impact its many neighbors (corresponding to many different geographic regions).

Given complete freedom in placing  $E = k * N$  edges on  $N$  nodes, it is possible to construct a topology that has a minimum vertex-cut of  $2k$ . In other words, the  $E$  edges can be placed in such a way that even in the presence of any  $2k - 1$  node failures in the graph, the resulting topology will still remain connected. We term such a placement of edges that maximizes the size of the vertex cut as an *optimal placement*. In the optimal placement, all the vertices have the same degree, viz.  $2 * k$ . For the simple case of  $k = 1$ , the optimal placement results in a ring topology. Although this optimal placement may be difficult to construct due to practical constraints, it provides us a nice reference point for comparing the fault tolerance of ISP topologies. In order to contrast an ISP's topology from the optimal scenario, we look at the degree distribution of the nodes. We say that a graph has a *skewed* degree distribution if its node degrees are distributed over a wide range with a few large node degrees and a high percentage of the nodes are leaves. The Internet topology exhibits a skewed degree distribution which can be characterized by a power law as described in [4].

Among the 9 commercial ISPs, some of them such as AT&T and Genuity have a very skewed degree distributions while other ISPs such as PSINet and Verio have much less skewed degree distributions (closer to optimal). The degree distribution will not be affected much due to a few missing links. Figure 15 shows the degree distributions of AT&T and PSINet. AT&T's topology has the maximum percentage of leaves among the 9





**Figure 15: Degree Distribution of Geographic Topologies of ISPs**

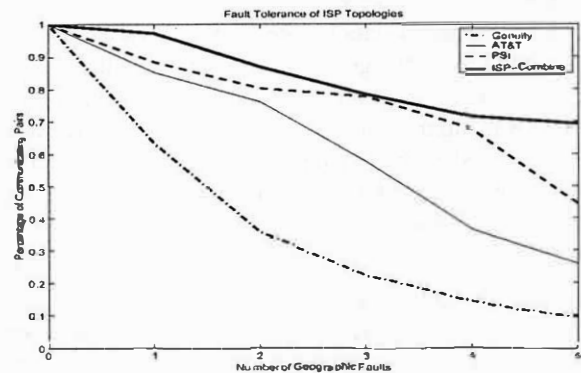
ISP topologies (62%) and has a few nodes with a degree greater than 12 (Chicago, Dallas). On the other hand, more than 50% of PSINet's nodes have a degree of either 2 or 3. This matches the optimal degree for Verio given that it has an edge to node ratio  $k = 1.5$ , which corresponds to an optimal degree of  $2 * k = 3$ . The ISP-Combine curve shows the degree distribution of the geographic topology obtained by combining the topology graphs of all 9 ISPs. The geographic nodes corresponding to the same city in the individual ISP topologies map to a single node in the combined topology. The combined topology still has a significant skew in its degree distribution. 29% of the nodes continue to be leaves. This happens despite the combined topology having an edge to node ratio of  $k = 2.5$ , which corresponds to an optimal degree of 5. On the other hand, nodes located in the important networking hubs of U.S. (e.g, San Jose, Washington DC, Chicago) have a degree of more than 20 in the combined topology.

## 6.2 Failure of high connectivity nodes

The skewed degree distributions of many tier-1 ISPs indicate that many geographic regions of an ISP may get disconnected if some high connectivity geographic nodes fail. To evaluate this, we consider the failure scenario where the  $f$  nodes of highest degrees in a graph fail.

We define a pair of geographic nodes that are connected by a network path and can communicate with each other as a *communicating pair*. A connected topology of  $N$  nodes can support  $N(N + 1)/2$  communicating pairs. (Since each node represents a geographic region, we also consider intra-node communication of a node with itself.) Under the scenario where the  $f$  nodes of highest degrees fail, the graph is disconnected into a forest where a node can only communicate with other nodes in its connected component. A connected component with

$m < N$  nodes can support  $m * (m + 1)/2$  communicating pairs. In the simple case where the parent of a leaf node fails, it produces a connected component of size 1 which supports exactly one communicating pair.



**Figure 16: Tolerance to Geographic Failures**

Figure 16 shows the percentage of communicating pairs supported in the various ISP networks in face of a varying number of geographic failures. The combined topology of the 9 ISPs supports 68% of the communicating pairs even after the removal of 5 important networking hubs in the US (San Jose, New York, Washington DC, Chicago, Los Angeles). Among the 9 ISPs, while Genuity and PSINet exhibit the least and the best fault tolerance characteristics. In the face of a single node failure, most of the ISPs lose between 15% and 30% of their communicating pairs in the worst case.

It is important to note that these results may represent a near-worst case failure scenario for the ISPs. If, however, many backup links are missing from our topology, the fraction of communicating pairs may be much higher than what we have portrayed. However, our essential message from this analysis is that a balanced degree distribution is a good feature for building a fault tolerant topology for an ISP.

## 7 Conclusions

In this paper, we have presented geography as a means for analyzing various aspects of Internet routing. First, our analysis based on extensive traceroute data shows the existence of many circuitous routes in the Internet. From the end-to-end perspective, we observe that the circuitousness of routes depends on the geographic and network locations of the end-hosts. We also find that the minimum delay along a path is more strongly correlated with the linearized distance the path than it is with the geographic distance between the end-points. This suggests that the circuitousness of a path does impact its minimum delay characteristics, which is an important end-to-end performance metric. In ongoing work, we are

studying the correlation between geography and network performance.

Second, a more careful examination shows that many circuitous paths tend to traverse multiple major ISPs. Although many of these major ISPs have points of presence in common locations, the peering between them is restricted to specific geographic locations, which causes the paths traversing multiple ISPs to be more circuitous. We also found that intra-ISP paths are far less circuitous than inter-ISP paths. An important requirement to reduce the circuitousness of paths is for ISPs to have peering relationships at many geographic locations.

Third, the fraction of the end-to-end path that lies within an ISP's network varies widely from one ISP to another. Furthermore, when we consider paths that traverse two or more major ISPs, we find that the path generally traverses a significantly shorter distance in the first ISP's network than in the second. This finding is consistent with the hot-potato routing policy. Using geographic information, we are able to quantify the degree to which an ISP's routing policy resembles hot-potato routing.

Finally, our analysis of geographic fault tolerance of ISPs indicates that the (IP-level) network topologies of many tier-1 ISPs exhibit skewed degree distributions which may induce a low tolerance to the failure of a single, critical geographic node. The combined topology of multiple ISPs exhibits better fault tolerance characteristics, assuming that the ISPs peer at all geographic locations that are in common.

## Acknowledgments

Vern Paxson made his 1995 data set available to us. Arvind Arasu, B. R. Badrinath, Mary Baker, Paul Barford, John Byers, Imrich Chlamtac, Mike Dahlin, Kevin Jeffay, Craig Labovitz, Paul Leyland, Karthik Mahesh, Vijay Parthasarathy, Jerry Prince, Amin Vahdat, Srinivasan Venkatachary, Geoff Voelker, Marcel Waldvogel, and David Wood helped us obtain access to a geographically distributed set of measurement hosts. Vern Paxson and the anonymous USENIX reviewers provided useful comments on an earlier version of this paper. We would like to thank them all.

## References

- [1] D. G. Andersen, H. Balakrishnan, R. Morris, and F. Kaashoek. Resilient Overlay Networks, *ACM SOSR*, November 2001.
- [2] B. Cheswick, H. Burch, and S. Branigan. Mapping and Visualizing the Internet, *USENIX Technical Conference*, June 2000.
- [3] C. Diot. Personal communication, November 2001.
- [4] M. Faloutsos, P. Faloutsos and C. Faloutsos. On Power-Law Relationships of the Internet Topology. *ACM SIGCOMM*, August 1999.
- [5] L. Gao. On Inferring Autonomous System Relationships in the Internet. *IEEE Global Internet*, November 2000.
- [6] R. Govindan and H. Tangmunarunkit, Heuristics for Internet Map Discovery. *IEEE Infocom*, March 2000.
- [7] K. Harrenstien, M. Stahl, and E. Feinler, NICK-NAME/WHOIS, *RFC-954, IETF*, October 1985.
- [8] V. Jacobson, Traceroute software, 1989, <ftp://ftp.ee.lbl.gov/traceroute.tar.gz>
- [9] C. Labovitz, J. Malan, and F. Jahanian. Internet Routing Instability. *ACM SIGCOMM*, August 1997.
- [10] B. Lyles. Personal communication, August 2001.
- [11] D. Moore et.al. Where in the World is net-geo.caida.org? *INET 2000*, June 2000.
- [12] J. Moy. OSPF Version 2. *RFC-2328, IETF*, April 1998.
- [13] V. N. Padmanabhan and L. Subramanian. An Investigation of Geographic Mapping Techniques for Internet Hosts. *ACM SIGCOMM*, August 2001.
- [14] V. Paxson. Measurements and Analysis of End-to-End Internet Dynamics. Ph.D. dissertation, UC Berkeley, 1997. <ftp://ftp.ee.lbl.gov/papers/vp-thesis/dis.ps.gz>
- [15] C. Semeria. Traffic Engineering for the New Public Network. Juniper Networks Whitepaper, September 2000.
- [16] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). *RFC-1771, IETF*, March 1995.
- [17] S. Savage, A. Collins, E. Hoffman, J. Snell and T. Anderson. The End-to-end Effects of Internet Path Selection, *ACM SIGCOMM*, pp 289-299, September, 1999.
- [18] <http://geography.about.com/>
- [19] Internet2. <http://www.internet2.org/>
- [20] Internet Traffic Archive. <http://ita.ee.lbl.gov/>

- [21] List of Public Libraries in the U.S.  
*<http://sunsite.berkeley.edu/Libweb>*
- [22] List of Public Traceroute Servers  
*<http://www.traceroute.org/>*
- [23] List of Web servers in Europe *<http://pauli.uni-muenster.de/w3world/Europe.html>*
- [24] MapNet: Macroscopic Internet Visualization and Measurement.  
*<http://www.caida.org/tools/visualization/mapnet/>*
- [25] Matrix.Net, *<http://www.matrix.net>*
- [26] NPD-Routes data set, Internet Traffic Archive.  
*<http://ita.ee.lbl.gov/html/contrib/NPD-Routes.html>*
- [27] Traceroute data used in this paper.  
*<http://sahara.cs.berkeley.edu/rawtraces>*
- [28] Skitter project at CAIDA.  
*<http://www.caida.org/tools/measurement/skitter/>*
- [29] vBNS: very high performance Backbone Network Service. *<http://www.vbns.net/>*
- [30] VisualRoute, Visualware Inc.  
*<http://www.visualroute.com/>*



# Providing Process Origin Information to Aid in Network Traceback

Florian P. Buchholz

*CERIAS*

*Purdue University*

florian@cerias.purdue.edu

Clay Shields

*Department of Computer Science*

*Georgetown University*

clay@cs.georgetown.edu

## Abstract

It is desirable to hold network attackers accountable for their actions in both criminal investigations and information warfare situations. Currently, attackers are able to hide their location effectively by creating a chain of connections through a series of hosts. This method is effective because current host audit systems do not maintain enough information to allow association of incoming and outgoing network connections. In this paper, we introduce an inexpensive method that allows both on-line and forensic matching of incoming and outgoing network traffic. Our method associates origin information with each process in the system process table, and enhances the audit information by logging the origin and destination of network sockets. We present implementation results and show that our method can effectively record origin information about the common cases of stepping stone connections and denial of service zombies, and describe the limitations of our approach.

## 1 Introduction

As the Internet has become a widely accepted part of the communications infrastructure there has been an increase in the number of network attacks [18]. One factor in the growth of attacks is that network attackers are only rarely caught and held accountable for their actions, giving them relative impunity in action. This situation has arisen, in part, because of the relative ease that attackers have in hiding their location, making it difficult and expensive for investigators to determine the origin of an attack.

In general, attackers use two different methods to hide their location [16]. One method, common in denial-of-service attacks, is to spoof the source address in IP packet headers so that recipients cannot easily determine the true source. As discussed further below, this has been an area of significant research in recent years. The other method, which has received significantly less attention from the research community, is for attackers to sequentially log into a number of (typically compromised) hosts. These forwarding hosts, often called *stepping-stone* hosts [40], effectively disguise the origin of the connection, as each host on the path sees only the previous host on the *connection chain*. A victim of an attack would not be able to determine the source of an attack without tracing the path back through all intermediate stepping-stone hosts. The audit data currently maintained at hosts is generally insufficient to correlate incoming and outgoing network traffic, so research about this problem has concentrated only on what can be deduced from network-level data. However, streams can be modified or delayed in a host so that a correlation is no longer possible from a network-level point of view, necessitating a host-based solution.

In this paper we will discuss a simple and inexpensive method for maintaining the necessary information to correlate data entering a host with data leaving a host. The goal of this work is to provide additional audit data that can help determine the source of network attacks. We include results from an implementation for the FreeBSD 4.1 kernel that show the technique is effective in providing information useful in tracing common attack situations, particularly for tracing stepping stones and denial-of-service attack zombies.



The next section provides a complete background of related work in the area and provides our view of the problem and design criteria to be addressed in providing a solution. Section 3 describes the technique we use to obtain and maintain location information for each process, and the logging mechanisms that can be used to provide forensic access to the data. Section 4 describes the specific application of the technique to the FreeBSD 4.1 kernel, and is followed in Section 5 by examples of the implementation in action. Section 6 outlines the limitations of our approach, and how these limitations can be addressed in future work. Finally, Section 7 provides a summary of our work.

## 2 Background

The goal of *network traceback* research is to allow determination of the source of attack traffic, so that a particular host used by a human to initiate an attack can be identified, and real-world investigative techniques used to locate the person responsible.

In order to accomplish this, the two problems described above — locating the source of IP packets and determining the first node of a connection chain — need to be solved. As described below, there has been significant research in locating the source of IP packets, and there have been efforts made to identify connection chains sources by examining network traffic. What is lacking is a reliable method of correlating incoming network traffic to a host with outgoing network traffic emanating from the host. This paper presents a mechanism for doing this. While our method is not always reliable, as discussed in Section 6, we believe that with further research and community involvement, this work can help address what is a serious problem.

### 2.1 Packet Source Determination

In normal operation, a host receiving packets can determine their source by direct examination of the source address field in the IP packet header. Unfortunately, this address is easy to falsify, making it simple for attackers to send packets that have their source effectively hidden. This is more common for one-way communication, such as the UDP and ICMP packets used in denial-of-service

attacks, but has been of use in attacks using TCP streams [21, 2]. There has been significant recent research in how to locate the source of such packets, primarily motivated by distributed denial-of-service (DDoS) attacks in early February of 2000. While it is generally recommended that routers be configured to perform ingress or egress routing [11], it is clear from continuing denial-of-service attacks [20] that this is not widely done. There have been other methods proposed to perform filtering to limit the effect of such attacks [24, 14].

As it is currently not possible to prevent such attacks, recent work has focused on how to locate the source of attacks. Some methods add or collect information at routers to allow traceback of DoS traffic [6, 27, 35, 7, 30]. Other methods add markings to the packets to probabilistically allow determination of the source given sufficient packets [28, 31, 23, 8, 9], or forward copies of packets, encapsulated in ICMP control messages, directly to the destination [1, 37]. A more innovative method uses counter-DoS attacks to locate the source of on-going attacks [4]. While we do not require that these schemes be available, we can make effective use of the traceback information they provide.

### 2.2 Correlating Streams

Research addressing determination of the source of a connection chain has mainly focused on correlating streams of TCP connections observed at different points in the network. Figure 1 shows an example of a connection chain.

The initial work in matching streams constructed *thumbprints* of each stream based on content [32]. While this technique could effectively match streams, it would be ineffective in compressed or encrypted streams as are common today. Other work compared the rate of sequence number increase in TCP streams as a matching mechanism, which can work as long as the data is not compressed at different hops and does not see excessive network delay [38]. Another technique, which relies solely on the timing of packets in a stream, is effective against encrypted or compressed streams of interactive user data [40]. This work was originally intended for intru-

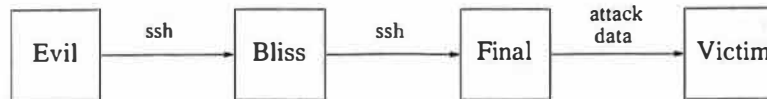


Figure 1: A sample connection chain

sion detection purposes but was also proposed as an effective method for finding the source of connection chains. While performing stream matching might be effective in some cases, such methods rely on examining network information, and might be vulnerable to the same methods that can be used to defeat network intrusion detection systems [26].

### 2.3 Forensic System Analysis

One of the objectives of computer forensics is the reconstruction of events that occurred on a system. Tools like the *Coroner's Toolkit* [10] attempt to discover hidden and deleted files and use access times to deduce system activities. A more formal model for file and event reconstruction is given by Andrew Gross [13].

In order to solve the host causality problem using forensic tools, it is necessary that network traffic is logged in some fashion on the host. Usually the essential information needed to associate incoming and outgoing traffic is not provided as the default on a system. While tools like TCP wrappers do a fine job logging incoming network traffic for the essential services, usually outgoing traffic is not logged at all. Furthermore, at best network activity can be tied to a particular user on the system. Exactly which processes and programs are involved may be obscured if there is a large amount of activity by that user.

### 2.4 Host causality

Though certain aspects of the network traceback problem have been addressed by the approaches described above, a new area of research that is concerned with data transformations or data flow tracking through a host is needed for a complete picture for attack origin traceback. We call this new area *host causality*, because we are attempting to

determine what network input causes other network output.

Common operating systems do not currently provide information that can match incoming and outgoing network traffic. While there has been some work that attempts to use existing system information to match active incoming and outgoing streams [15, 5], this work has been either shown to be impractical to securely implement [3], or requires an external trigger to store forensic information. Ideally, it should be possible to determine whether network traffic was originated directly from a particular host, or occurred as a result of a connection from some other remote machine, and, if possible, which remote machine is involved. This would not only help in tracing back to the source of a network attack, but could be useful in showing due diligence, so that the owner of a machine used in attack could demonstrate that the attack originated elsewhere.

A solution that addresses the problem of tracing connections through a host is necessary because a host on the network can transform data passing through it in such a way that, from the network's point of view, it can no longer be easily related to traffic leaving it. This might be the case in a stepping stone scenario, if the traffic is delayed, or differently compressed or encrypted. Also, in attacks like a distributed denial-of-service (DDoS) attack [39], control traffic cannot be linked to the resulting attack traffic. In such an attack, packet source-location techniques might identify the source of a particular attack stream, but will not allow identification of the master or the controlling host. This is due to the fact that the datagrams that are used to perform the attack are seemingly unrelated to those that control the client. What is missing is information within the host that can be used to associate an incoming control packet with outgoing attack data.

### 2.4.1 Desired Properties

The following properties either need to be fulfilled or seem desirable in order to achieve a practical solution to the host causality problem:

1. It must be possible to determine whether a given process on the host was started by a local user or remotely.
2. If a process was started by a user at a remote location, information about that source must be maintained and associated with the process.
3. An audit facility must exist that allows the logging of incoming network traffic and processes that receive it. This will allow correlation between the source of a process and the source of incoming network packets.
4. An audit facility must exist that allows the logging of individual outgoing network traffic and processes that send it. Combined with the facility above, one could then relate incoming and outgoing traffic processed by the same process.
5. The logs maintained about origin information should be resistant to modifications by attackers.
6. Processes that spawn other processes need to pass on their source information to their children, or, if they provide a remote login service, pass on the remote location as the child's new source.
7. The modifications to a system should be minimal so that they do not interfere with existing software.
8. Due to restricted logging space, it should be possible to use rules to control what data the audit system collects.
9. It should be possible to quickly identify processes that were not started locally together with their remote location.

## 3 Description of Model

A process on a computing host is an executing instance of a program [34]. Processes are therefore, among many other things, responsible for receiving and generating network data on a host that is connected to a network.

Processes can be started:

- explicitly by a human being
- by the system

A human being can start processes:

- while physically present at the host
- from a remote location
- indirectly through some other process he or she started

The system can start processes:

- through startup scripts (including `init`)
- through scheduling services like `cron` and `at`
- through system services like `inetd`

The *origin of a process* is the information about how any process running on the system was started in regard to the above possibilities. For the purpose of this paper only a distinction between a process that was started by a human being from a remote location (*remote origin*) and the other ways (*local origin*) is of importance, with the exception of the special case of indirectly started processes.

In case of a remote origin for a process, the origin information should include that remote location. If the system tracks the origin of a process and a process sends out network traffic and is of remote origin, then the system can make a connection between the traffic that was sent out and the traffic that was received from the origin of the process over the network. The traffic could be individual datagrams, or they can be part of an established connection.

In order to gain access to a system from a remote location and start new processes there, a user has to make use of a service offered on that particular system. Usually most systems provide well-known services such as `telnet`, `rsh`, or `ssh` that will give a remote user a shell on the system. However, there are other possibilities to create new processes that do not involve an interactive shell. In fact, any process listening on an open port on the system may be used or misused for such purposes. Our solution does not address these problems, and they are a topic for future investigation.

As the only legitimate remote access to a system is through its well-known services, it is feasible to store information about the existing connection with the newly created child process. After a successful login procedure, the source of the new process should reflect the information stored about the connection. Note that the origin of a process and its subsequent children is set at the time a user gains access to the system. All programs that will be started during that remote session will inherit that origin. At this point time delays become irrelevant, as origin information is stored with the processes no matter whether or not processes become dormant for any amount of time.

### 3.1 Information Storage

From the viewpoint of a host, all that can be deduced about the origin of an arriving network packet is the interface that it arrived on and the information that is contained in the packet itself. A host on its own cannot determine whether a network datagram was spoofed or not. Therefore, for IP packets, the five-tuple consisting of source and destination IP addresses and source and destination ports and the protocol number must suffice to distinguish source information maintained about processes on a host. If packet traceback schemes are deployed and can provide additional information, it is possible to maintain that information as well.

While storing information about active processes can be useful, for complete analysis of attacks, some additional information needs to be logged as well. The logging mechanism can maintain more explicit information than simply storing the IP five-tuples. Along with the five-tuple and timestamp, the system can also store the interface on which the packet arrives and the process id. If the system logs individual packets, it can also store a checksum of the non-changing parts of each packet header that is logged in case the need for a more detailed post-analysis matching arises.

Furthermore, it would be expensive and impractical to log an entire stream of packets that make up the entirety of a TCP stream. Since TCP is a connection oriented transport layer protocol, it is sufficient to only regard incoming and outgoing SYN requests

for the purposes of logging. Unfortunately, UDP is a connection-less protocol. Thus for UDP, all packets need to be logged. Log-reducing mechanisms that group the same kind of UDP packets together can certainly be applied here, but this is out of the scope of this paper.

### 3.2 Limitations on Information Availability

For well-known services we can assume that there will only be one open network connection for each child process spawned as they adhere to the common style of running Unix servers that fork for each new request [33]. Non-standard server programs might behave differently, however, and there might be multiple open connections when we try to determine the origin information. In this case, it is impossible to be sure which connection should be considered as the origin of the process. Because of this, there can be a problem with using the latest data from the accept system call as the origin information. If a server program allows multiple open sockets before the call to *login*, then there is a possibility that the wrong origin information is stored with the process. It is possible to design a program that after accepting a connection opens another listening socket to receive a decoy connection from a completely different remote site or, even worse, from the local host itself. This would set the information obtained from the accept call to the new socket's source data, before *login* was invoked. After a successful login procedure, the origin information would be incorrect. If a local user installs such a program, then any attacks originating from it can be viewed as originating from the host, which is consistent with our definition of local origin.

Another problem is that a remote user may still hide his real origin by creating a connection from the system to itself. In this case the origin information of the process gets changed to the source information of the local host. While the process is still being considered of remote origin, it is of no value from a traceback perspective. If many remote processes "change" their origin in such a fashion, one cannot determine anymore what the "real" origin of any of those was. In order to prevent this obscuring of the origin of a process,

one needs to keep track of an *inheritance line* for remote processes. That is, for any given process of remote origin, one must be able to determine its parent process if that parent process also was of remote origin.

## 4 Implementation

The model described above was implemented in the FreeBSD 4.1 operating system on an i386 based PC. While the implementation is therefore specific to the UNIX operating system, the general principles of the model should be applicable to other systems as well.

All processes that accept network connections do need to make use of the socket system calls provided by the system. Stevens [33] describes the necessary steps to set up a TCP or UDP server. They involve system calls to `bind`, `listen`, and `accept`, in that order. Thus any connection between two systems must have successfully undergone a call to `accept` on the server side. In the case of TCP, `accept` returns after a successful three-way handshake. In the case of UDP, `accept` returns upon reception of a packet that matches the socket characteristics.

As a successful connection implies a successful return from the `accept` system call, it seems reasonable to make modifications there in order to obtain location information. Specifically, with the assumption of only one open network connection, it is sufficient to record the data from the last call to `accept`. This information will then be accessible to the child process created by the `fork` system call. Finally, after a successful login procedure, the source of the new process should reflect the information stored about the connection. As the login program lies in user space and not all well-known servers utilize it, it will be necessary to perform this step through one of the system calls such as `setlogin`.

All the necessary information described in Section 3 is available within data structures used by the `accept` system call. Once the connection has been established, the socket descriptor contains the source IP address and the source port of the purported source of the traffic. To determine the destination IP

address and port that was used to establish the connection, the system also has to access the protocol control block (PCB) that is associated with the socket and that is pointed to from the socket data structure<sup>1</sup>. The information can be obtained through simple pointer lookups.

### 4.1 Where to store source information

We decided to maintain the information directly in the process table itself, because it is simple to add another field that contains the necessary information, and creation and termination of processes is handled automatically. The inheritance problem is taken care of as well, as the `fork` system call causes certain fields of the process table to be copied to the child. The only time we therefore need to access the field in the process table is when origin information changes. The disadvantage of this approach is that some auxiliary programs such as `top` and `ps` might have to be adjusted to accommodate the changes.

It is possible to utilize existing logging facilities, such as `syslog` to record the data, or a logging program can develop its own format and location to store the information [25]. Ideally, there would be some mechanism to ensure the integrity of the logs. Write-once, read-many media, or a secure logging facility could be used [29].

### 4.2 Data structures and kernel modifications

For the source information, a new data type, `struct porigin`, was created as shown in Figure 2.

The `type` field denotes whether the source is local (0) or remote (1). If the type is 0, all other fields are undefined and can be ignored. The next five fields are the typical four-tuple for a TCP or UDP connection, consisting of source and destination IP addresses as well as source and destination ports, plus the protocol number. The last parameter is a timestamp, which denotes the time the connection was established in network time format [19]. Note that the network interface is

<sup>1</sup>See McKusick et al. [17] or Wright and Stevens [36] for further details.



```

struct porigin {
    char          type;
    struct in_addr source_ip;
    struct in_addr dst_ip;
    u_short       source_port;
    u_short       dst_port;
    u_short       proto;
    time_t        tstamp;
};

```

Figure 2: The process origin data structure

not included here but can be obtained with the information stored if necessary.

In order to keep track of the corresponding source information for each process, the process table data structure (`struct proc`) was modified in two locations. It is necessary to retain the actual source information as well as information about the last accepted connection of a process. The latter is needed because all common TCP/IP based network services that provide a remote login facility first accept the connection and then fork off a child process where *login* is called.

Hence, two fields, `origin` and `lastaccept` were added to the process table structure, both of type `struct porigin`. The fields are located in the area that gets copied in the fork system call.

The copying of the `origin` field provides a simple and elegant solution for the inheritance mechanism. All it takes is a few more bytes to be copied in the fork system call, as the process structure is copied anyway. Thus, a child process always inherits the source information from its parent.

This leaves the question of where the two fields, `lastaccept` and `origin` are to be set. As the name already suggests, `lastaccept` is set in the `accept` system call, after a successful accept of an incoming connection. The modified `accept` system call was implemented as shown in Figure 3, which shows how to retrieve information from the PCB.

Note that `accept1` is called from the actual `accept` system call. The connection will be accepted in the procedure `soaccept`. If the call is successful, the type is set to 1, and the four-tuple is obtained from the PCB associated with the socket via the pointer `inp`.

Note that this will only work for a TCP connection, which is used by services which provide a shell. For future work, other protocol types need to be considered. For instance, in the case of UDP, the `recvfrom` system call may be modified in a similar fashion.

The `origin` field will be copied from a parent process to its child. However, as discussed above, each time a *login* is performed within a process, the source information of the last accepted socket should become the new origin information for that process. Thus, at an invocation of *login*, the `lastaccept` field should be copied into the `origin` field. However, as discussed above, *login* is only a program in user space that simply utilizes several system calls to perform the actual user login. One could supply a separate system call to have the `lastaccept` field copied to the `origin` field, but that would imply that every program that supplies a login service to be changed and use it. Therefore, one of the system calls used by every login service, `setlogin` was modified so that the field is copied after a successful call.

To keep track of the inheritance line for a remote process, it is necessary to modify the fork system call, as well. It is sufficient to record the process IDs of the parent and child processes in case the parent is of remote origin. From this information, it is possible to reconstruct the entire inheritance line for a remote process up to the first parent that was of remote origin. The `syslog` facility provides an easy way to log kernel messages, and was chosen to record the information out of reasons of simplicity. Figure 4 shows the modifications made to `fork`. In future work, this recording mechanism needs to be refined and optimized.

```

accept1(p, uap, compat)
{
    (void) soaccept(so, &sa);
    inp = sotoinpcb(so); // pointer to protocol control block
    populate fields in p->lastaccept from information
        pointed to by inp;
    p->lastaccept.type = 1;
}

```

Figure 3: The modified accept system call (pseudo-code)

### 4.3 System calls

In order to access the source information for a given process, a new system call, `getorigin` was added to the system. It takes as parameters a process identifier and a buffer, into which the source information is copied.

Note that there is no system call to set or reset the origin field. With the `getorigin` system call, it is now possible to design logging facilities and administrative programs within user space that make use of the source information of a process. For reasons of simplicity, the call was implemented to be unrestricted.

Another system call, `portpid`, was added to give support for the logging facility described below. If one wants to associate incoming TCP or UDP packets with the receiving process, one needs to find the process id of the socket that will handle an IP packet. The same is true for sockets that are responsible for outgoing packets. Those sockets are identified in the network layer by the four-tuple of source and destination addresses and ports, but, unfortunately, there is no mechanism in FreeBSD to obtain that information within user space. Thus the system call `portpid` will take such a four-tuple as well as a protocol identifier (TCP or UDP) and will return the process id of the process that belongs to the listening socket that will accept packets matching the four-tuple, or belonging to the socket that sent the packet. If there is no such socket, an error will be returned. A weakness of this design is that a process may exit and be removed from the process table before the `portpid` call occurs. More verbose logging could offset this problem.

The FreeBSD operating system uses *protocol control blocks* (PCBs) to demultiplex incoming IP packets. The PCBs are chained together in a linked list and contain IP source and destination addresses and TCP or UDP ports or wildcard entries for incoming packets to match against. Each PCB also contains a pointer to the socket that is destined to receive a packet, should it match the four-tuple specified in the PCB. From the socket, one can then look in the receive or write buffer to obtain the actual process id of the receiving or sending process, respectively. In order to determine which process will receive a packet or which process sent a packet, one needs to traverse the list of PCBs until the best match is found, and then obtain the process id of the socket associated with the PCB.

### 4.4 Logging facility

The logging facility that was implemented is merely a proof of concept, and there are many feasible ways to design and implement one. Our implementation of the logging facility uses the `libpcap` library, which is part of the Berkeley Packet Filter (BPF). The BPF will make a copy of each incoming and outgoing network packet that matches given filter criteria and supply that copy to the process utilizing the filter.

This prototype logging facility can therefore be considered as a network sniffer, but a more robust and efficient implementation would be one that is part of the kernel itself. For each TCP SYN or UDP packet seen by the sniffer, the `portpid` system call is invoked to obtain the process id of the process responsible for the packet. Once the process id is obtained, `getorigin` is called for that process id to determine whether the process is of remote origin or not. If it is of remote origin, then the packet as well as the origin

```

int
fork(p, uap) {
    ...
    error = fork1(p, RFFDG | RFPROC, &p2);
    if (error == 0) {
        p->p_retval[0] = p2->p_pid;
        p->p_retval[1] = 0;
        if (p->origin.type)
            log(LOG_INFO,
               "remote process %d spawned child %d\n",
               p->p_pid, p2->p_pid);
    }
    ...
}

```

Figure 4: The modified fork system call

information is printed out. Figure 5(a) shows the interaction of the different parts of the system with the logging facility, and Figure 6 shows the important parts of the routine that processes the packets passed on by the BPF.

There is a problem with logging outgoing UDP packets. The `portpid` system call relies on the socket that sent the packet to be still open so that it can find it in the PCB list. If an application opened a socket, wrote one UDP packet, and immediately closed the socket again, there is a chance that the socket no longer exists when the packet is examined by the logging facility. DDoS clients usually keep the socket they send packets from open so that packets can be sent at a faster rate, but for outgoing control packets, this is a problem. For TCP, this is not a serious issue, as there is either a three-way handshake or a time-wait period at the end of each connection.

One method to solve the outgoing UDP packet problem could entail further modification of the kernel, keeping the process ids of sending processes in a cache and making that information available to the `portpid` system call. A similar approach could also improve lookup performance for incoming packets. Instead of duplicating the de-multiplexing effort made in the networking stack, modifications to the stack could result in a new data structure that returns the correct process id for a given five-tuple.

## 5 Implementation Results

The modified kernel was installed on an Intel Pentium III 866 MHz Celeron PC. The machine used is part of a small networking

lab. We will discuss the effects of the changes on the normal system behavior as well as give two examples of processes of remote origin handling traffic.

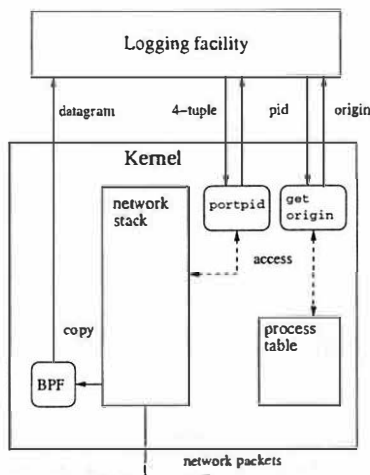
### 5.1 Effects on normal system behavior

As the changes to the system were only few and cheap, the impact on the system is minimal. The `getorigin` copies a few bytes from the process table and is only executed for TCP SYN and UDP packets. For those packets, the call to `portpid` causes a linked-list traversal of the protocol control blocks in the same manner the networking stack does its de-multiplexing. In every call to `accept`, the `lastaccept` field is set from the socket information. These operations are very few and inexpensive compared to the entire set of operations within `accept`. In every call to `fork`, an extra few bytes need to be copied to pass on the origin information to a process's child. The way the `syslog` facility was used to keep record of an inheritance line is very inefficient. On a system where processes spawn many children, the logs may quickly wrap around. That and the fact that the inheritance line needs to be reconstructed manually from the logs suggests the need for a redesign of the inheritance line for future work.

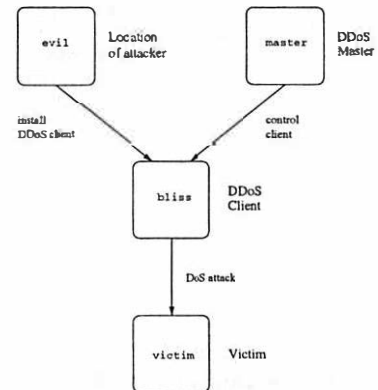
### 5.2 Examples

#### 5.2.1 Stepping Stone

In this example, `bliss` was used as a stepping stone. A user from `evil` (10.0.0.1) logged into `bliss` (192.168.0.1) via `ssh`. From there, he used `ssh` again, to log into `final` (172.16.0.1). The actual host names and IP addresses have been replaced by fictitious



(a) The logging facility



(b) Setup for the DDoS attack

Figure 5: The logging facility and DDoS attack experimental setup

ones. This setup is equal to the example given in Figure 1 with the exception of the very last host.

The logging facility recorded the following entry from this:

```
192.168.0.1:1022->172.16.0.1:22 sent by pid 285
Origin: 10.0.0.1:1022-192.168.0.1:22
```

One can observe, that the origin information indicates the connection from evil, port 1022, to bliss, port 22 (ssh). The logging mechanism didn't log the connection from evil to bliss, as sshd is a local process. However, evil is clearly shown as the origin for the process that connected to final. Therefore one can now associate the stream from bliss to final to the one from evil to bliss for traceback purposes.

### 5.2.2 DDoS Client

In this example, a DDoS trinoo client, obtained from the Packet Storm archive [22], was installed on bliss from evil. The corresponding master was installed on another machine, master (192.168.0.2). Bliss was then used via master to perform a denial of service attack against victim (172.16.0.2), a third machine in the test network. Figure 5(b) shows the setup for the attack.

Again, host names and IP addresses have been changed.

A sample of the logging output is presented in Figure 7.

The first logged event is a UDP packet from bliss to master, notifying the trinoo master that a client is active. The next event is then a UDP packet from master to bliss, triggering the DoS attack. The rest of the log shows UDP packets sent from bliss to victim as part of the attack.

All the traffic can be unambiguously associated with the process 3760, the DDoS client. From the origin, one can see that the process was started from evil. In this example, it is clear that the attack was controlled from master. This might not always be possible, as multiple packets from different locations could be received by the process just before an attack. However, by examining the logs a good estimate might be derived. At the very least it will give a list of possible hosts from where the attack was launched. Network traceback mechanisms can now be used to determine the location from where the software was set up and master could now be investigated in the same manner as bliss to determine more information about the attack.

```

if (protocol is TCP) {
    set pointer to TCP header within the packet;
    remember source and destination ports;
    if (this is the start of a new connection)
        set log flag;
}
else if (protocol is UDP) {
    set pointer to UDP header within the packet;
    remember source and destination ports;
    set log flag;
}
if (log flag is set) {
    if (packet is coming in)
        invoke portpid with parameters for incoming packets;
    else if (packet is going out)
        invoke portpid with parameters for outgoing packets;
    else
        set error;

    if (portpid returned successfully) {
        call getorigin with pid returned by portpid;
        if (origin is remote) {
            if (packet is coming in)
                print log for incoming packets;
            else if (packet is going out)
                print log for outgoing packets;
        }
    }
}

```

Figure 6: The packet processing routine of the logging facility (pseudo-code)

```

192.168.0.1:1117->192.168.0.2:31335 (17) sent by pid 3760
Origin: 10.0.0.1:32155-192.168.0.1:13419

192.168.0.2:39805->192.168.0.1:27444 (17) received by pid 3760
Origin: 10.0.0.1:32155-192.168.0.1:13419

192.168.0.1:1135->172.16.0.2:12865 (17) sent by pid 3760
Origin: 10.0.0.1:32155-192.168.0.1:13419
192.168.0.1:1135->172.16.0.2:59850 (17) sent by pid 3760
Origin: 10.0.0.1:32155-192.168.0.1:13419
192.168.0.1:1135->172.16.0.2:10435 (17) sent by pid 3760
Origin: 10.0.0.1:32155-192.168.0.1:13419
192.168.0.1:1135->172.16.0.2:4577 (17) sent by pid 3760
Origin: 10.0.0.1:32155-192.168.0.1:13419

```

Figure 7: Output of the logging facility

and the location of the attacker.

## 6 Limitations and Future Work

This paper presents a first attempt at a mechanism designed to address the problem of determining host causality. While it is progress in a forward direction, it is not a complete solution to a problem, though its use could prove beneficial in many cases. We hope that discussion of the limitations will foster other research on the problem.

While available origin information is maintained for processes that utilize `setlogin`, there are other mechanisms that attackers can use to start processes on a system. Remotely, attackers might gain access to a system using processes that service network requests, such as mail, web, or ftp servers. Exploits such as buffer overflows against these

processes can produce user shells for the attacker, bypassing the system call. In these cases, origin information will not be properly recorded. For these cases the question arises when exactly to set the origin information so that it is meaningful. Furthermore, an attacker who gains access to a system might use a cron or at job to create a process after the attacker has logged off; this would also result in processes that lack the correct origin information. A solution to this problem might be to include origin information in the file system so that when the new process was started the appropriate location information was available.

Sometimes login servers can open a second connection to the client for out-of-band data. Currently this scenario is not handled in the design. However, it seems that in the worst



case the wrong port is recorded for the origin within the modified `accept` system call.

An attacker also might use a covert channel between processes to obscure the proper location information. In this scenario, an attacker, who perhaps enters the system through a mechanism that invokes `setlogin` and whose processes therefore have correct origin information, uses some form of IPC to cause a process that has other origin information to send data into the network. This is a difficult problem to deal with, as it has always been [12], and we do not have an immediate solution for it. Any process that listens on a covert channel needs to have been started either locally or remotely, however, and in the case of an external attacker, most likely remotely. Thus, any outgoing traffic from that process will still be logged.

While our implementation only operates on TCP and UDP packets, any protocol could be used by an attacker. For example, some DDoS tools use ICMP messages to send control messages over the network. In this case, an attacker would either have to modify the routines for ICMP processing in the kernel or may have to sniff the incoming traffic using a library like `libpcap`. If the attacker has modified the kernel to listen to and process these messages, there seems to be little that can be done to establish the origin information for a process, because if the kernel can be modified by the attacker, the origin information can be tampered with as well. In the latter case one can check for open BPF filters and also be aware of processes that utilize other protocols or do not receive network packets from the networking stack but rather through the packet filter.

The mechanism for keeping track of the inheritance line for a process needs to be improved. The current mechanism, while very simple to implement, is the only part of the modifications we made that affects the system in a noticeable fashion. One problem is that with each new child process, more information needs to be stored, even though it is small. Once a separate data structure for keeping inheritance lines is used, a simple improvement would be to delete inheritance lines or parts of it where all the processes

involved have terminated. However, overall management of the inheritance lines remains as future work.

In the event of a system compromise, in which an attacker gains root capabilities, the origin information in the kernel and recorded information in the file system is just as vulnerable to modification or deletion as any other kernel or file system information. We consider this outside of the scope of our work, but point to other work that attempts to make audit information survive such attacks [29], and suggest that current forensics tools could be modified to recover the altered origin information in some cases.

Finally, as mentioned above, the packet logging system is a prototype only; a more effective design would be to include the logging mechanism in the kernel itself. Instead of sniffing for outgoing packets, writes to a network socket would cause the outgoing packet to be logged before the socket could be closed, alleviating the problem with trying to find the source of UDP packets mentioned above. Additionally, the current mechanism logs all TCP SYN and UDP packets, creating a denial-of-service opportunity for attackers to fill up disk space, so a more selective approach to recording packets is clearly in order, where possible.

## 6.1 Future work in Host Causality

Even though the origin information was designed with network traceback in mind, there are other applications or foundations for new modifications of the system:

- A system administrator can use the origin information to determine the origins of all running processes and identify ones that have a very unusual source. This can lead to the discovery of running DDoS clients on a machine, for example.
- The origin information can be incorporated into the file system. By storing a process's origin information with a file whenever the process writes to the file system. Not only can this help in solving the problem with `cron` and startup scripts, but it can also aid in locating suspicious programs in users' home directories. This would be especially ef-

fective with logging file systems, so that the changes in files could be tracked by location as well.

- Origin information adds another dimension to access control. Access control mechanisms can be altered so that they take origin information into account and grant certain privileges only when certain origin conditions are met.
- Statistics based on origin of processes can be gathered, which can be used to profile normal system behavior or to locate trends that may help in better system administration.

Origin information may well benefit in other security related fields. The prospect of access control in combination with origin information seems to be an especially interesting area. Research in that direction may well improve overall robustness of the origin mechanism itself.

## 7 Conclusion

In this paper, we have introduced the notion of host causality as a mechanism to complement current research in network traceback. With the addition of origin information to a process, we have developed a mechanism that, with only minor changes to the given system, works well under the simple circumstances. The two examples show that important information for network traceback can be obtained with origin information and the new logging possibilities that result from that.

The work we presented here is only the start of work in the overall area. We have identified many limitations of our mechanism, and outlined what future work needs to be done to better address the problem. Host causality is not a complete solution to all the problems that faced in tracing connections through a network, but providing solutions could prove a valuable tool to help improve security in a future networking environment.

## References

- [1] S. Bellovin. ICMP Traceback Messages. Technical report, IETF Internet draft, March 2000. Work in progress.
- [2] S. M. Bellovin. Security Problems in the TCP-IP Protocol Suite. *Computer Communications Review*, 19(2):32–48, April 1989.
- [3] F. Buchholz, T. Daniels, B. Kuperman, and C. Shields. Packet tracker final report. Technical Report 2000-23, CERIAS, Purdue University, 2000.
- [4] H. Burch and B. Cheswick. Tracing Anonymous Packets to their Approximate Source. In *Proceedings of the 14th Conference on Systems Administration (LISA-2000)*, New Orleans, LA, December 2000.
- [5] B. Carrier and C. Shields. A Recursive Session Token Protocol for use in Computer Forensics and TCP Traceback. In *Proceedings of the IEEE Infocomm 2002*, 2002. To appear.
- [6] H. Chang and D. Drew. DoSTracker. This was a publicly available PERL script that attempted to trace a denial-of-service attack through a series of Cisco routers. It was released into the public domain, but later withdrawn., June 1997.
- [7] Characterizing and Tracing Packet Floods Using Cisco Routers. <http://www.cisco.com/warp/public/707/22.html>.
- [8] D. Dean, M. Franklin, and A. Stubblefield. An Algebraic Approach to IP traceback. In *Proceedings of the 2001 Network and Distributed System Security Symposium*, San Diego, CA, February 2001.
- [9] T. W. Doepfner, P. N. Klein, and A. Koyfman. Using Router Stamping to Identify the Source of IP Packets. In *7th ACM Conference on Computer and Communications Security*, pages 184–189, Athens, Greece, November 2000.
- [10] D. Farmer and W. Venema. The Coroner's Toolkit (TCT). <http://www.fish.com/tct/>.
- [11] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. Technical Report RFC 2827, Internet Society, May 2000.
- [12] Virgil Gligor. A guide to understanding covert channel analysis of trusted systems. Technical Report NCSC-TG-030, National Computer Security Center, Ft. George G. Meade, Maryland, U.S.A., November 1993. Approved for public release: distribution unlimited.
- [13] A. H. Gross. *Analyzing Computer Intrusions*. PhD thesis, University of California, San Diego, 1997.

- [14] J. Ioannidis and S. M. Bellovin. Pushback: Router-Based Defense Against DDoS Attacks. In *Proceedings of the 2002 Network and Distributed System Security Symposium*, San Diego, CA, February 2002.
- [15] H. T. Jung, H. L. Kim, Y. M. Seo, G. Choe, S. L. Min, C. S. Kim, and K. Koh. Caller Identification System in the Internet Environment. In *UNIX Security Symposium IV Proceedings*, pages 69–78, 1993.
- [16] S. C. Lee and C. Shields. Tracing the Source of Network Attack: A Technical, Legal, and Societal Problem. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, West Point, NY, June 2001.
- [17] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, Boston, MA, 1996.
- [18] B. McWilliams. CERT: Cyber Attacks Set To Double In 2001. <http://www.securifyfocus.com/news/266>.
- [19] D.L. Mills. Network Time Protocol. RFC 1059, July 1988.
- [20] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial of Service Activity. In *Proceedings of the 2001 USENIX Security Symposium*, Washington D.C., August 2001.
- [21] R.T. Morris. A Weakness in the 4.2BSD Unix TCP-IP Software. Technical Report 17, AT&T Bell Laboratories, 1985. Computing Science Technical Report.
- [22] Distributed Attack Tools Section. <http://packetstorm.securify.com/distributed/>.
- [23] K. Park and H. Lee. On the effectiveness of probabilistic packet marking for IP traceback under denial of service attack. In *Proceedings IEEE INFOCOM 2001*, pages 338–347, April 2001.
- [24] K. Park and H. Lee. On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets. In *Proceedings of the 2001 ACM SIGCOMM*, San Diego, CA, August 2001. To Appear.
- [25] J. Picciotto. The Design of An Effective Auditing Subsystem. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 13–22, 1987.
- [26] T. Ptacek and T. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.
- [27] J. Rowe. Intrusion Detection and Isolation Protocol: Automated Response to Attacks. Presentation at Recent Advances in Intrusion Detection (RAID), 1999.
- [28] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, August 2000.
- [29] B. Schneier and J. Kelsey. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security*, 1(3), 1999.
- [30] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, and W. T. Strayer S. T. Kent. Hash-Based IP Traceback. In *Proceedings of the 2001 ACM SIGCOMM*, San Diego, CA, August 2001.
- [31] D. X. Song and A. Perrig. Advanced and Authenticated Marking Schemes for IP Traceback. In *Proceedings of the IEEE Infocomm 2001*, April 2001.
- [32] S. Staniford-Chen and L.T. Heberlein. Holding Intruders Accountable on the Internet. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 39–49, Oakland, CA, May 1995.
- [33] W. R. Stevens. *Unix Network Programming*, volume 1. Prentice Hall PTR, second edition, 1998.
- [34] W.R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, MA, 1993.
- [35] R. Stone. CenterTrack: An IP Overlay Network for Tracking DoS Floods. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.
- [36] G.R. Wright and W.R. Stevens. *TCP/IP Illustrated Volume 2, The Implementation*. Addison Wesley, Boston, MA, 1995.
- [37] S. F. Wu, L. Zhang, D. Massey, and A. Mankin. Intention-Driven ICMP Traceback. IETF Internet draft, February 2001. Work in progress.
- [38] K. Yoda and H. Etoh. Finding a Connection Chain for Tracing Intruders. In *Proceedings of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*, October 2000.
- [39] ZDNet Special Report: It's War! Web Under Attack. <http://www.zdnet.com/zdnn/special/doswebattack.html>, February 2000.
- [40] Y. Zhang and V. Paxson. Detecting Stepping Stones. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

# Cyclone: A safe dialect of C

Trevor Jim\*

Greg Morrisett†

Dan Grossman†

Michael Hicks†

James Cheney†

Yanling Wang†

## Abstract

Cyclone is a safe dialect of C. It has been designed from the ground up to prevent the buffer overflows, format string attacks, and memory management errors that are common in C programs, while retaining C's syntax and semantics. This paper examines safety violations enabled by C's design, and shows how Cyclone avoids them, without giving up C's hallmark control over low-level details such as data representation and memory management.

## 1 Introduction

It is a commonly held belief in the security community that safety violations such as buffer overflows are unprofessional and even downright sloppy. This recent quote [33] is typical:

Common errors that cause vulnerabilities — buffer overflows, poor handling of unexpected types and amounts of data — are well understood. Unfortunately, features still seem to be valued more highly among manufacturers than reliability.

The implication is that safety violations can be prevented just by changing priorities.

It's true that highly trained and motivated programmers can produce extremely robust systems when security is a top priority (witness OpenBSD). It's also true that most programmers can and should do more to ensure the safety and security of the programs that they write. However, we believe that the reasons that safety violations show up so often in C

programs reach deeper than just poor training and effort: they have their roots in the design of C itself.

Take buffer overflows, for example. Every introductory C programming course warns against them and teaches techniques to avoid them, yet they continue to be announced in security bulletins every week. There are reasons for this that are more fundamental than poor training:

- One cause of buffer overflows in C is bad pointer arithmetic, and arithmetic is tricky. To put it plainly, an off-by-one error can cause a buffer overflow, and we will never be able to train programmers to the point where off-by-one errors are completely eliminated.
- C uses NUL-terminated strings. This is crucial for efficiency (a buffer can be allocated once and used to hold many different strings of different lengths before deallocation), but there is always a danger of overwriting the NUL terminator, usually leading to a buffer overflow in a library function. Some library functions (`strcat`) have alternate versions (`strncat`) that help, by letting the programmer give a bound on the length of a string argument, but there are many dozens of functions in POSIX with no such alternative.
- Out-of-bounds pointers are commonplace in C. The standard way to iterate over the elements of an array is to start with a pointer to the first element and increment it until it is just past the end of the array. This is blessed by the C standard, which states that the address just past the end of any array must be valid. When out-of-bounds pointers are common, you have to expect that occasionally one will be dereferenced or assigned, causing a buffer overflow.

In short, the design of the C programming language encourages programming at the edge of safety. This makes programs efficient but also vulnerable, and leads us to conclude that safety violations are likely

\*AT&T Labs Research, [trevor@research.att.com](mailto:trevor@research.att.com)

†Cornell University, <http://www.cs.cornell.edu/projects/cyclone>

to remain common in C programs. A number of studies bear this out [23, 11, 28, 18].

If C programs are unsafe, it is tempting to suggest that all programs be written in a safe language like Java (or ML, or Modula-3, or even 40-year-old Lisp). However, this is not a realistic solution for everyone. For one thing, it abandons legacy code. For another, all of the safe languages look very different from C: they are high-level and abstract, they do not have explicit memory management, and they do not give programmers control over low-level data representations. These features make C unique, efficient, and indispensable to systems programmers.

We are developing an alternative for those who want safety but do not want to switch to a high-level language: Cyclone, a dialect of C that has been designed to prevent safety violations. Our goal is to design Cyclone so that it has the safety guarantee of Java (no valid program can commit a safety violation) while keeping C's syntax, types, semantics, and idioms intact. In Cyclone, as in C, programmers can "feel the bits." We think that C programmers will have little trouble adapting to our dialect and will find Cyclone to be an appropriate language for many of the problems that ask for a C solution.

Cyclone has been in development for two years. In total, we have written about 110,000 lines of Cyclone code, with about 35,000 lines for the compiler itself, and 15,000 lines for supporting libraries and tools, like a port of the Bison parser generator. We have also ported about 50,000 lines of benchmark applications, and are developing a streaming media overlay network in Cyclone [27]. Cyclone is freely available and comes with extensive documentation. The compiler and most of the accompanying tools are licensed under the GNU General Public License, and most of the libraries are licensed under the GNU LGPL.

This paper is a high-level overview of Cyclone. It presents the design philosophy behind Cyclone, gives an overview of the techniques we've used to make a safe version of C, and reviews the history of the project, the mistakes we've made, and the course corrections that they inspired.

The remainder of the paper is organized as follows. Section 2 points out some of the features of C that can lead to safety violations, and describes the changes we made to prevent this in Cyclone. Section 3 gives some details about our implementation

and its performance. Section 4 discusses the evolution of Cyclone's design, pointing out key decisions that we made and mistakes that we later reversed. We discuss future work in Section 5. In section 6, we discuss existing approaches to making C safer, and explain how Cyclone's approach is different. We conclude in Section 7.

## 2 From C to Cyclone

Most of Cyclone's language design comes directly from C. Cyclone uses the C preprocessor, and, with few exceptions, follows C's lexical conventions and grammar. Cyclone has pointers, arrays, structures, unions, enumerations, and all of the usual floating point and integer types; and they have the same data representation in Cyclone as in C. Cyclone's standard library supports a large (and growing) subset of POSIX. The intention is to make it easy for C programmers to learn Cyclone, to port C code to Cyclone, and to interface C code with Cyclone code.

The major differences between Cyclone and C are all related to safety. The Cyclone compiler performs a static analysis on source code, and inserts run-time checks into the compiled output at places where the analysis cannot determine that an operation is safe. The compiler may also refuse to compile a program. This may be because the program is truly unsafe, or may be because the static analysis is not able to guarantee that the program is safe, even by inserting run-time checks. We reject some programs that a C compiler would happily compile: this includes all of the unsafe C programs as well as some perfectly safe programs. We must reject some safe programs, because it is impossible to implement an analysis that perfectly separates the safe programs from the unsafe programs.

When Cyclone rejects a safe C program, the programmer may choose to rewrite the program so that our analysis can verify its safety. To make this easier, we have identified common C idioms that our static analysis cannot handle, and have added features to the language so that these idioms can be programmed in Cyclone with only a few modifications. These modifications typically include adding annotations that supply hints to the static analysis, or that cause the program to maintain extra information needed for run-time checks (e.g., bounds checks).

Table 1: Restrictions imposed by Cyclone to preserve safety

- NULL checks are inserted to prevent segmentation faults
  - Pointer arithmetic is restricted
  - Pointers must be initialized before use
  - Dangling pointers are prevented through region analysis and limitations on `free`
  - Only “safe” casts and unions are allowed
  - `goto` into scopes is disallowed
  - `switch` labels in different scopes are disallowed
  - Pointer-returning functions must execute `return`
  - `setjmp` and `longjmp` are not supported
- 

Cyclone can thus be understood by starting from C, imposing some restrictions to preserve safety, and adding features to regain common programming idioms in a safe way. Cyclone’s restrictions are summarized in Table 1, and its extensions are summarized in Table 2.

Some of the techniques we use to make Cyclone safe have been applied to C before, and there has been a great deal of research on additional techniques that we do not use in Cyclone. However, previous projects have typically used only one or two techniques, resulting in incomplete coverage. For example, McGary’s bounded pointers protect against some, but not all, array access violations [26], and StackGuard protects against some, but not all, buffer overflows [9]. Our goal with Cyclone is to prevent all safety violations. Moreover, previous projects have been presented as optional add-ons to C, so in practice they are seldom used in production code; Cyclone makes safety the default.

In the rest of this section, we illustrate Cyclone’s features by giving examples of safety violations in C code, explaining how Cyclone’s restrictions detect and prevent them, and introducing the language extensions that can be used to safely program around the restrictions. Some of the safety violations we describe, like buffer overflows, can lead to root exploits. All of them can lead to crashes, which can be exploited to mount denial of service

Table 2: Extensions provided by Cyclone to safely regain C programming idioms

- Never-NULL pointers do not require NULL checks
  - “Fat” pointers support pointer arithmetic with run-time bounds checking
  - Growable regions support a form of safe manual memory management
  - Tagged unions support type-varying arguments
  - Injections help automate the use of tagged unions for programmers
  - Polymorphism replaces some uses of `void *`
  - Varargs are implemented with fat pointers
  - Exceptions replace some uses of `setjmp` and `longjmp`
- 

attacks [6, 7, 12, 15, 25, 16].

NULL Consider the `getc` function:

```
int getc(FILE *);
```

If you call `getc(NULL)`, what happens? The C standard gives no definitive answer. If `getc` is written with safety in mind, it will perform a NULL check on its argument. That would be inefficient in the common case, though, so the check is probably omitted, leading to a segmentation fault.

Cyclone provides two solutions. The first is to automatically insert run-time NULL checks when pointers are used. For example, Cyclone will insert code into the body of `getc` to do a NULL check when its argument is dereferenced.

This requires little effort from the programmer, but the NULL checks slow down `getc`. To repair this, we have extended Cyclone with a new kind of pointer, called a “never-NULL” pointer, and indicated with ‘@’ instead of ‘\*’. For example, in Cyclone you can declare

```
int getc(FILE @);
```



indicating that `getc` expects a non-NULL FILE pointer as its argument. This one-character change tells Cyclone that it does not need to insert NULL checks into the *body* of `getc`. If `getc` is called with a possibly-NULL pointer, Cyclone will insert a NULL check at the *call*:

```
extern FILE *f;
getc(f);           // NULL check here
```

Cyclone prints a warning when it inserts the NULL check. This can be suppressed with an explicit cast:

```
getc((FILE @)f); // Check w/o warning
```

A programmer can force the NULL check to occur only once by declaring a new @-pointer variable, and using the new variable at each call:

```
FILE @g = (FILE @)f; // NULL check here
getc(g);             // No NULL check
```

Finally, constants like `stdin` are declared as @-pointers in the first place, and functions can be declared to return @-pointers. The effect is that NULL checks can be pushed back from their uses all the way to their sources. This is just as in C, except that in Cyclone, the compiler can ensure that NULL dereferences do not occur.

Never-NULL pointers are a perfect example of Cyclone's design philosophy: safety is guaranteed, automatically if possible, and the programmer has control over where any needed checks are performed.

**Buffer overflows** To prevent buffer overflows, we restrict pointer arithmetic: Cyclone does not permit pointer arithmetic on \*-pointers or @-pointers. Instead, we provide another kind of pointer, indicated by '?', which permits pointer arithmetic. A ?-pointer is represented by an address plus bounds information; since the representation of a ?-pointer takes up more space than a \*-pointer or @-pointer, we call it a "fat" pointer. The extra information in a fat pointer allows Cyclone to determine the size of the array pointed to, and to insert bounds checks at pointer accesses to ensure safety.

Here's an example of fat pointers in use — the string length function written in Cyclone:

```
int strlen(const char ?s) {
    int i, n;
    if (!s) return 0;
    n = s.size;
    for (i = 0; i < n; i++, s++)
        if (!*s) return i;
    return n;
}
```

This looks like a C version of `strlen`, with two exceptions. First, we declare the argument `s` to be a fat pointer to `char`, rather than a \*-pointer. Second, in the body of the function we are able to get the size of the array pointed to by `s`, using the notation `s.size`. This lets us check that `s` is in-bounds in the for loop. That means we are guaranteed that we will never dereference `s` outside the bounds of the string, even if the NUL terminator is missing. In contrast, the C `strlen` will scan past the end of a string that lacks a NUL terminator.

Fat pointers add overhead to programs, because they take up more space than other pointers, and because of inserted bounds checks. However, they ensure safety, they give the programmer new capabilities (finding the size of the base array), and the programmer has explicit control over where they are used. It's easy to use ?-pointers in Cyclone. A programmer who wants to use a ?-pointer only needs to change a single character ('\*' to '?') in a declaration. Arrays and strings are converted to ?-pointers as necessary (automatically by the compiler). A programmer can explicitly cast a ?-pointer to a \*-pointer (this inserts a bounds check) or to a @-pointer (this inserts a NULL check and a bounds check). A \*-pointer or @-pointer can be cast to a ?-pointer, without any checks; the resulting ?-pointer has size 1.

**Uninitialized pointers** The following snippet of C crashed one author's Palm Pilot:

```
Form *f;
switch (event->eType) {
case frmOpenEvent:
    f = FrmGetActiveForm(); ...
case ctlSelectEvent:
    i = FrmGetObjectIndex(f, field); ...
}
```

This is part of a function that processes events. The problem is that while the pointer `f` is properly ini-

tialized in the first case of the `switch`, it is (by oversight) not initialized in the second case. So when the function `FrmGetObjectIndex` dereferences `f`, it isn't accessing a valid pointer, but rather an unpredictable address — whatever was on the stack when the space for `f` was allocated.

To prevent this in Cyclone, we perform a static analysis on the source code. The analysis detects that `f` might be uninitialized in the second case, and the compiler signals an error. Usually, this catches a real bug, but there are times when our analysis isn't smart enough to figure out that something is properly initialized. This may force the programmer to initialize variables earlier than in C.

We don't consider it an error if non-pointers are uninitialized. For example, if you declare a local array of non-pointers, you can use it without initializing the elements:

```
char buf[64];    // contains garbage ..
sprintf(buf,"a"); // .. but no err here
char c = buf[20]; // .. or even here
```

This is common in C code; since these array accesses are in-bounds, we allow them.

**Dangling pointers** Here is a naive (unsafe!) version of a C function that takes an `int` and returns its string representation:

```
char *itoa(int i) {
    char buf[20];
    sprintf(buf,"%d",i);
    return buf;
}
```

The function allocates a character buffer on the stack, prints the `int` into the buffer, and returns a pointer to the buffer. The problem is that the caller now has a pointer into deallocated stack space; this can easily lead to safety violations.

It is easy for a C compiler to warn against returning the address of a local variable, and, indeed, `gcc` prints just such a warning for the example above. However, this technique will not catch even the following simple variation:

```
char *itoa(int i) {
```

```
    char buf[20];
    char *z;
    sprintf(buf,"%d",i);
    z = buf;
    return z;
}
```

Here, the address of `buf` is stored in the variable `z`, and then `z` is returned. This passes `gcc -Wall` without complaint.

Cyclone prevents the dereference of dangling pointers by performing a *region analysis* on the code. A region is a segment of memory that is deallocated all at once. For example, Cyclone considers all of the local variables of a block to be in the same region, which is deallocated on exit from the block. Cyclone's static region analysis keeps track of what region each pointer points into, and what regions are live at any point in the program. Any dereference of a pointer into a non-live region is reported as a compile-time error.

In this last example, Cyclone's region analysis knows that the address of `buf` is a pointer into the local stack of `itoa`. The assignment to `z` tells Cyclone that `z` is also a pointer into `itoa`'s stack area. Since the local stack area will be deallocated when `z` is returned from `itoa`, we report an error.

Cyclone's region analysis is intraprocedural — it is not a whole-program analysis. We rely on programmer annotations to track regions across function calls. For example, the `strcat` function is declared as follows in Cyclone:

```
char ?'r strcat(char ?'r dest,
                const char ? src);
```

Here `'r` is a *region variable*. The declaration says that for any region `'r`, `strcat` takes a pointer `dest` into region `'r`, and a pointer `src`, and returns a pointer into region `'r`. (In fact, the C standard specifies that `strcat` returns `dest`.) This information enables Cyclone to correctly reject the following program:

```
char ?itoa(int i) {
    char buf[20];
    sprintf(buf,"%d",i);
    return strcat(buf, "");
}
```

The region analysis deduces that the result of the call to `strcat` on `buf` points into the local stack region of `itoa`, so it cannot be returned from the function.

Cyclone's region analysis is described in greater detail in a separate paper [21].

**Free** C's `free` function can create dangling pointers, and, depending on how it is implemented, can cause segmentation faults or even root compromises if used incorrectly (e.g., if it is called with a pointer not returned by `malloc` [16], or if it is used to reclaim the same block of memory twice [7]). It is difficult to design an analysis that can guarantee the correct use of pointers and `free`, so our current solution is drastic: we make `free` a no-op.

Obviously, programmers still need a way to reclaim heap-allocated data. We provide two ways. First, the programmer can use an optional garbage collector. This is very helpful in getting existing C programs to port to Cyclone without many changes. However, in many cases it constitutes an unacceptable loss of control.

We recognize that C programmers need explicit control over allocation and deallocation. Therefore, Cyclone provides a feature called *growable regions*. The following code declares a growable region, does some allocation into the region, and deallocates the region:

```
region h {
    int *x = rmalloc(h, sizeof(int));
    int ?y = rnew(h) { 1, 2, 3 };
    char ?z = rprintf(h, "hello");
}
```

The code uses a `region` block to start a new, growable region that lives on the heap. The region is deallocated on exit from the block (without an explicit `free`). The variable `h` is a *handle* for the region and it is used to allocate into the region, in one of several ways.

First, there is an `rmalloc` construct that behaves like `malloc` except that it requires a region handle as an argument; it allocates into the region of the handle. In the example above, `x` is initialized with a pointer to an `int`-sized chunk of memory allocated in `h`'s region.

Second, the `rnew` construct is used when the programmer wants to allocate and initialize in a single step. For example, `y` is initialized above as a fat pointer to an array with elements 1, 2, and 3, allocated in `h`'s region.

Finally, region handles may be passed to functions like the library function `rprintf`. `rprintf` is like `sprintf`, except that it does not print to a fixed-sized buffer; instead it allocates a buffer in a region, places the formatted output in the buffer, and returns a pointer to the buffer. In the example above, `z` is initialized with a pointer to the string "hello" that is allocated in `h`'s region. Unlike `sprintf`, there is no risk of a buffer overflow, and unlike `snprintf`, there is no risk of passing a buffer that is too small. Moreover, the allocated buffer will be freed when the region goes out of scope, just as a stack-allocated buffer would be.

Our region analysis knows that `x`, `y`, and `z` all point into `h`'s region, and that the region is deallocated on exit from the block. It uses this knowledge to prevent dangling pointers into the region — for example, it prohibits storing `x` into a global variable, which could be used to (wrongly) access the region after it is deallocated.

Growable regions are a safe version of arena-style memory management, which is widely used (e.g., in Apache). C programmers use many other styles of memory management, and we plan in the future to extend Cyclone to accommodate more of them safely. In the meantime, Cyclone is one of the very few safe languages that supports safe, explicit memory management, without relying on a garbage collector.

**Type-varying arguments** In C it is possible to write a function that takes an argument whose type varies from call to call. The `printf` function is a familiar example:

```
printf("%d", 3); printf("%s", "hello");
```

In the first call to `printf`, the second argument is an `int`, and in the next call, the second argument is a `char *`. This is perfectly safe in this case, and the compiler can even catch errors by examining the format string to see what types the remaining arguments should have. Unfortunately, the compiler can't catch all errors. Consider:

```
extern char *y; printf(y);
```

This is a lazy way to print the string `y`. The problem is that, in general, `y` can contain `%` format directives, causing `printf` to look for non-existent arguments on the stack. The compiler can't check this because `y` is not a string literal. A core dump is not unlikely.

The danger is greater if the user of the program gets to choose the string `y`. The `%n` format directive causes `printf` to write the number of characters printed so far into a location specified by a pointer argument; it can be used to write an arbitrary value to a location chosen by the attacker, leading to a complete compromise. This is known as a format string attack, and it is an increasingly common exploit [34].

We solve this in Cyclone in two steps. First, we add *tagged unions* to the language:

```
tunion t {
    Int(int);
    Str(char ?);
};
```

This declares a new tagged union type, `tunion t`. A tagged union has several cases, like an ordinary union, but adds tags that distinguish the cases. Here, `tunion t` has an `int` case with tag `Int`, and a `char ?` case with tag `Str`. A function that takes a tagged union as argument can look at the tags to find out what case the argument is in, using an extension of the `switch` statement:

```
void pr(tunion t x) {
    switch (x) {
        case &Int(i): printf("%d", i); break;
        case &Str(s): printf("%s", s); break;
    }
}
```

The first case of the `switch` will be executed if `x` has tag `Int`; the variable `i` gets bound to the underlying `int`, so it can be used in the body of the case. Similarly, the second case is taken if `x` has tag `Str` with underlying string `s`.

Tags enable the `pr` function above to correctly detect the type of its argument. However, callers have to explicitly add tags to the arguments. For example, `pr` can be called as follows:

```
pr(new Int(4));
pr(new Str("hello"));
```

The first line calls `pr` with the `int` 4, adding the tag `Int` with the notation `new Int(4)`. The second call does the same with string "hello" and tag `Str`.

Inserting the tags by hand is inconvenient, so we also provide a second feature, *automatic tag injection*. For example, in Cyclone, `printf` is declared

```
printf(char ?fmt, ... inject parg_t);
```

where `parg_t` is a tagged union containing all of the possible types of arguments for `printf`. Cyclone's `printf` is called just as in C, without explicit tags:

```
printf("%s %i", "hello", 4);
```

The compiler inserts the correct tags automatically (they are placed on the stack). The `printf` function itself accesses the tagged arguments through a fat pointer (Cyclone's varargs are bounds checked) and uses `switch` to make sure the arguments have the right type. This makes `printf` safe even if the format string argument comes from user input — Cyclone does not permit the `printf` programmer to use the arguments in a type-inconsistent way. Moreover, the tags let the programmer detect any inconsistency at run time and take appropriate action (e.g., return an error code or exit the program).

Type-varying arguments are used in many other POSIX functions, including the `scanf` functions, `fcntl`, `ioctl`, `signal`, and socket functions such as `bind` and `connect`. Cyclone uses tagged unions and injection to make sure that these functions are called safely, while presenting the programmer with the same interface as in C.

**Goto** C's `goto` statements can lead to safety violations when they are used to jump into scopes. Here is a simple example:

```
int z;
{ int x = 0xBAD; goto L; }
{ int *y = &z;
  L: *y = 3;    // Possible segfault
}
```

The program declares a variable `z`, then enters two blocks in sequence. Many compilers stack allocate the local variables of a block when it is entered, and deallocate (pop) the storage when the block exits (though this is not mandated by the C standard). If the example is compiled in this way, then when the program enters the first block, space for `x` is allocated on the stack, and is initialized with the value `0xBAD`. The `goto` jumps into the middle of the second block, directly to the assignment to the contents of the pointer `y`. Since `y` is the first (only) variable declared in the second block, the assignment expects `y` to be at the top of the stack. Unfortunately, that's exactly where `x` was allocated, so the program tries to write to location `0xBAD`, probably triggering a segmentation fault.

Cyclone's static analysis detects this situation and signals an error. A `goto` that does not enter a scope is safe, and is allowed in Cyclone. We apply the same analysis to `switch` statements, which suffer from a similar vulnerability in C.

**Other vulnerabilities** These are only a few of the features of C that can be misused to cause safety violations. Other examples are: bad casts; `varargs` (as implemented in C); missing return statements; violations of `const` qualifiers; and improper use of unions. Cyclone's analysis restricts these features to prevent safety violations.

### 3 Implementation

The Cyclone compiler is implemented in approximately 35,000 lines of Cyclone. It consists of a parser, a static analysis phase, and a simple translator to C. We use `gcc` as a back end and have also experimented with using Microsoft Visual C++. We are able to use some existing tools (`gdb`, `flex`) and we ported others completely to Cyclone (`bison`). When a user compiles with garbage collection enabled, we use the Boehm-Demers-Weiser conservative garbage collector as an off-the-shelf component. We have also built some useful utilities, including a documentation generation tool and a memory profiler.

In order to get a rough idea of the current and potential performance of the language, we ported a selection of benchmarks from C to Cyclone. The

Program	LOC		diffs		
	C	Cyc	#	C %	? %
cacm	340	360	41	12%	0%
cfrac	4218	4215	134	3%	37%
finger	158	161	17	11%	12%
grobner	3260	3401	452	14%	24%
http_get	529	530	44	8%	45%
http_load	2072	2058	121	6%	24%
http_ping	1072	1082	33	3%	33%
http_post	607	609	51	8%	45%
matxmult	57	53	11	19%	9%
mini_httpd	3005	3027	266	9%	46%
ncompress	1964	1986	134	7%	25%
tile	1345	1365	148	11%	32%
total	18627	18847	1452	8%	31%

*regionized benchmarks*

cfrac	4218	4192	503	12%	9%
mini_httpd	3005	2986	531	18%	24%
total	7223	7178	1034	14%	16%

Table 3: Benchmark diffs

benchmarks were useful in testing Cyclone's safety guarantees as well as its performance: several of the benchmarks had safety violations that were revealed (and we subsequently fixed) when we ported them to Cyclone. The process of porting also tested the limitations of Cyclone's interface to the C library and forced us to provide more complete library support. For example, even small benchmarks such as `finger` and `http_get` make use of parts of the C library that the Cyclone compiler and other tools do not, such as sockets and signals.

**The benchmarks** We tried to pick benchmarks from a range of problem domains. For networking, we used the `mini_httpd` web server; the web utilities `http_get`, `http_post`, `http_ping`, and `http_load`; and `finger`. The `cfrac`, `grobner`, `tile`, and `matxmult` benchmarks are computationally intensive C applications that make heavy use of arrays and pointers. Finally, `cacm` and `ncompress` are compression utilities. All of the benchmark programs, in both C and Cyclone, can be found on the Cyclone homepage [10].

**Ease of porting** We have tried to design Cyclone so that existing C code can be ported with few modifications. Table 3 quantifies the number of modifications we needed to port the benchmarks. For each benchmark, the table shows the number of lines of

code in both the C and Cyclone versions. The diff # column shows the number of lines changed in each port, and the C % column shows the percentage of lines changed relative to the original program size. In porting the first grouping of benchmarks, we tried to minimize changes. In particular, the benchmarks involving non-trivial dynamic memory management (*cfrac*, *grobner*, *http\_load*, and *tile*), were compiled with the garbage collector in Cyclone; all other benchmarks do not use the garbage collector. The second grouping gives results for versions of benchmarks that we modified to make use of Cyclone's growable regions wherever possible.

Usually fewer than 10% of the lines needed to be changed to port the benchmarks to Cyclone. One of the most common changes was changing C-style \* pointers to Cyclone ? pointers; for example, changing `char *` to `char ?`. The ? % column of Table 3 shows the percentage of changes that were of this form: generally, this simple change accounted for 20–50% of changed lines. Most of the other changes had to do with adapting to Cyclone's stricter requirements for allocation, initialization, `const` enforcement, and function prototyping. Typical changes of these forms included changing `malloc` to `new`, adding explicit initializers, adding explicit `const` type qualifiers to casts, and ensuring that all functions have prototypes with explicit return values.

**Performance** Table 4 compares the performance of the benchmarks in C, in Cyclone with bounds checking enabled, and in Cyclone with bounds checking disabled. Presently we do only very simple bounds-check elimination, because our effort to date has focused on safety, rather than performance; the gap between the second and third measurements gives an upper bound for the improvement we can expect from this in the future.

We ran each benchmark twenty-one times on a 750 MHz Pentium III with 256MB of RAM, running Linux kernel 2.2.16-12, using gcc 2.96 as a back end. We used the gcc flags `-O3` and `-march=i686` for compiling all the benchmarks. Because we observed skewed distributions for the *http* benchmarks, we report medians and semi-interquartile ranges (SIQR).<sup>1</sup> For the non-web benchmarks (and

<sup>1</sup>The semi-interquartile range is the difference between the high quartile and the low quartile divided by 2. This is a measure of variability, similar to standard deviation, recommended for skewed distributions [22].

some of the web benchmarks as well) the median and the mean were essentially identical, and the standard deviation was at most 2% of the mean.

The table also shows the slowdown factor of Cyclone relative to C. We achieve near-zero overhead for I/O bound applications such as the web server and the *http* programs, but there is a considerable overhead for computationally-intensive benchmarks; the worst is *grobner*, which is almost a factor of three slower than the C version. We have seen slowdowns of a factor of six in pathological scenarios involving pointer arithmetic in other microbenchmarks not listed here.

Two common sources of overhead in safe languages are garbage collection and bounds checking. The checked and unchecked columns of Table 4 show that bounds checks are an important component of our overhead, as expected. Garbage collection overhead is not as easy to measure. Profiling the garbage collected version of *cfrac* suggests that garbage collection accounts for approximately half of its overhead. Partially regionizing *cfrac* resulted in a 6% improvement with bounds checks on; but regionizing can require significant changes to the program, so the value of this comparison is not clear. We expect that the overhead will vary widely for different programs depending on their memory usage patterns; for example, *http\_load* and *tile* make relatively little use of dynamic allocation, so they have almost no garbage collection overhead.

Cyclone's representation of fat pointers turned out to be another important overhead. We represent fat pointers with three words: the base address, the bounds address, and the current pointer location (essentially the same representation used by McGary's bounded pointers [26]). Compared to C's pointers, fat pointers have a larger space overhead, larger cache footprint, increased parameter passing overhead, and increased register pressure, especially on the register-impovertished x86. Good code generation can make a big difference: we found that using gcc's `-march=i686` flag increased the speed of programs making heavy use of fat pointers (such as *cfrac* and *grobner*) by as much as a factor of two, because it causes gcc to use a more efficient implementation of block copy.

**Safety** We found array bounds violations in three benchmarks when we ported them from C to Cyclone: *mini\_httpd*, *grobner*, and *tile*. This was a



Test	C time(s)	Cyclone time			
		checked(s)	factor	unchecked(s)	factor
cacm	0.12 ± 0.00	0.15 ± 0.00	1.25×	0.14 ± 0.00	1.17×
cfrac <sup>†</sup>	2.30 ± 0.00	5.57 ± 0.01	2.42×	4.77 ± 0.01	2.07×
finger	0.54 ± 0.42	0.48 ± 0.15	0.89×	0.53 ± 0.16	0.98×
grobner <sup>†</sup>	0.03 ± 0.00	0.07 ± 0.00	2.85×	0.07 ± 0.00	2.49×
http.get	0.32 ± 0.03	0.33 ± 0.02	1.03×	0.32 ± 0.06	1.00×
http.load <sup>†</sup>	0.16 ± 0.00	0.16 ± 0.00	1.00×	0.16 ± 0.00	1.00×
http.ping	0.06 ± 0.02	0.06 ± 0.02	1.00×	0.06 ± 0.01	1.00×
http.post	0.04 ± 0.01	0.04 ± 0.00	1.00×	0.04 ± 0.01	1.00×
matxmult	1.37 ± 0.00	1.50 ± 0.00	1.09×	1.37 ± 0.00	1.00×
mini.httpd-1.15c	2.05 ± 0.00	2.09 ± 0.00	1.02×	2.09 ± 0.00	1.02×
ncompress-4.2.4	0.14 ± 0.01	0.19 ± 0.00	1.36×	0.18 ± 0.00	1.29×
tile <sup>†</sup>	0.44 ± 0.00	0.74 ± 0.00	1.68×	0.67 ± 0.00	1.52×

<sup>†</sup>Compiled with the garbage collector

regionized benchmarks					
cfrac	2.30 ± 0.00	5.22 ± 0.01	2.27×	4.55 ± 0.00	1.98×
mini.httpd-1.15c	2.05 ± 0.00	2.09 ± 0.00	1.02×	2.08 ± 0.00	1.01×

Table 4: Benchmark performance

surprise, since at least one (grobner) dates back to the mid 1980s. On the other hand, this is consistent with research that shows that such bugs can linger for years even in widely used software [28].

The `mini_httpd` web server consults a file, `.htpasswd`, to decide whether to grant client access to protected web pages. It tries to be careful not to reveal the password file to clients. Ironically, the code to protect the password file contains a safety violation:

```
#define AUTH_FILE ".htpasswd"
... strcmp(&(file[strlen(file) -
    sizeof(AUTH_FILE) + 1]),
    AUTH_FILE) == 0 ...
```

The code is trying to see if the file requested by the client is `.htpasswd`. Unfortunately, if `file` is a string shorter than `.htpasswd`, then `strcmp` will be passed an out-of-bounds pointer. This could result in access to `file` being denied (if the region of memory just before the string constant `".htpasswd"` happens to contain that file name), or it could cause the program to crash (if the region of memory is inaccessible). Cyclone found the error with a run-time bounds check.

The `grobner` benchmark had a more serious violation affecting both safety and correctness. The program represents polynomials as arrays of coeffi-

cients, and has a multiply routine that handles polynomials with a single coefficient as a special case. Unfortunately, the code for the general case turns out to be completely wrong: a loop is unrolled incorrectly, and the multiplication ends up being applied to out-of-bounds pointers. As a result, the answers returned are unpredictable. Four of the ten test cases provided in the distribution follow this code path (in our performance experiments above, we consider only the six correct input cases). In Cyclone, our bounds checks quickly illuminated the source of the problem.

The `tile` program had array bounds violations due to an off-by-one error and an order-of-evaluation bug in this code:

```
if (snum > cur_sentsize)
    mksentarrays(cur_sentsize,
        cur_sentsize += GROWSIZE);
```

The function `mksentarrays` reallocates several global arrays. Reallocation is supposed to occur when `snum` is greater than or equal to `cur_sentsize`; the if guard above has an off-by-one error. Cyclone caught this with a bounds check in `mksentarrays`. In addition, the first argument of `mksentarrays` should be the old size of the array, and the second argument should be the new size. Our platform uses right-to-left evaluation, so the code above passes the new size of the array to `mksentarrays` in *both* argu-

ments. Again, this was caught with a bounds check in Cyclone. In C, the out-of-bounds access was not caught, causing an incorrect initialization of the new arrays.

## 4 Design history

Cyclone began as an offshoot of the Typed Assembly Language (TAL) project [30, 20]. The TAL project's goal was to ensure program safety at the machine code level, by adding machine-checkable safety annotations to machine code. The machine code annotations are not easy to produce by hand, so we designed a simple, C-like language called Popcorn as a front end, and built a compiler that automatically translates Popcorn to machine code plus the necessary annotations.

Popcorn worked out well as a proof-of-concept for TAL, but it had some disadvantages. It was C-like, but different enough to make porting C code and interfacing to C code difficult. It was also a language that was used only by our own research group, and was unlikely to be adopted by anyone else. Cyclone is a reworking of Popcorn with two agendas: to further our understanding of low-level safety, and to gain outside adopters.

It turns out that taking C compatibility as a serious requirement was critical to advancing both of these agendas. It was obvious from the start that C compatibility would make Cyclone more appealing to others, but the idea that it would help us to understand how to better design a *safe* low-level language was a surprise.

C programmers don't write the same kinds of programs as programmers in safe languages like Java — they use many tricks that aren't available in high-level languages. While many C programs are not 100% safe, most are intended to be safe, and we learned a great deal from porting systems code from C to Cyclone. Often, we found that we had made choices in the design of Cyclone that were holdovers from ML [29], another language that we had worked on. Some (most!) of these choices were right for ML, but not for C, or for Cyclone, and we ended up following C more closely than we had expected at the start.

All of this has played out gradually over the years of

Cyclone's development. Here are some of the more notable mistakes and course changes we've made:

- Originally, we supported arrays not with fat pointers, but with a type `array<t>`, where `t` is the element type of the array. An `array<t>` could be passed to functions, and a value of type `array<t>` supported subscripting, but not pointer arithmetic. This matches up closely with ML's array types, and was a carryover from when Popcorn was implemented in ML. However, converting C code to use `array<t>` was painful, requiring nontrivial editing of type declarations, and converting pointer arithmetic to array subscripting. We abandoned it for fat pointers, which make it easy to port C code, requiring only a few changes from `*` to `?`, and no changes to pointer arithmetic.
- We didn't understand the importance of NUL-terminated strings. NUL termination isn't guaranteed in C, so, for safety, we were committed to using explicit array bounds from the beginning. The NUL seemed pointless, and our first string library ignored it. As we programmed more in the language and ported C code, we came to understand how important NUL is to efficiency (memory reuse), and we changed our string library to match up with C's.
- In C, a `switch` case by default falls through to the next case, unless there is an explicit `break`. This is exactly the opposite of what it should be: most cases do not fall through, and, moreover, when a case does fall through, it is probably a bug. Therefore, we added an explicit fallthru statement, and used the rule that a case would *not* fall through unless the fallthru statement was used.

Our decision to "correct" C's mistake was wrong. It made porting error-prone because we had to examine every `switch` statement to look for intentional fall throughs, and add a fallthru statement. We had also gotten rid of any special meaning of `break` within `switch`, since it was no longer needed — consequently, a `break` in a `switch` within a loop would break to the head of the loop (in early versions of Cyclone). Eventually, we realized that we were going against a basic instinct of every C programmer, without gaining much of anything, so we restored C's semantics of `switch` and `break`.

- We originally implemented tagged unions as an extension of enumerations, since an enumeration constant is like a case of a tagged union with no associated value. Since a tagged union is more general, we decided to just have one of the two.

This was a mistake because in C, an enumeration is really treated as `int`, and C programmers rely on this. It's not uncommon to see things like

```
x = (x+1)%3;
```

where `x` is an enumeration variable. We aren't able to do this with tagged unions, so we eventually separated them from `enum`.

## 5 Future work

C programmers use a wide variety of memory management strategies, but at the moment, Cyclone supports only garbage collection and arena memory management. A major goal of the project going forward will be to research ways to accommodate other memory management strategies, while retaining safety.

Another limitation of our current release is that assignments to fat pointers are not atomic, and hence, are not thread-safe. We plan to address this by requiring the programmer to acquire a lock before accessing a thread-shared fat pointer; this will be enforced by an extension of the type system. Locks will not be necessary for thread-local fat pointers.

We are experimenting with a number of new pointer representations. For instance, a pointer to a zero-terminated array can be safely represented as just an address, as long as the pointer only moves forward inside the array, and the zero terminator is never overridden. The new representations should make it easier to interface to legacy C code as well as improve on the space overhead of fat pointers.

Finally, we plan to explore ways to automatically translate C programs into Cyclone. We have the beginnings of this in the compiler itself (which tries to report informative errors at places where code needs to be modified), and in a tool we built to semi-automatically construct a Cyclone interface to C libraries.

## 6 Related work

There is an enormous body of research on making C safer. Most techniques can be grouped into one of the following strategies:

1. Static analysis. Programs like Lint crawl over C source code and flag possible safety violations, which the programmer can then review. Some other examples are LCLint [17, 24], Metal [13, 14], SLAM [3, 2], PREFIX [5], and cqual [32].
2. Inserting run-time checks. C's `assert` statements, the Safe-C system [1], and "debugging" versions of libraries, like Electric Fence, cause programs to perform sanity checks as they run. This technique has been used to combat buffer overflows [9, 4, 19] and `printf` format string attacks [8].
3. Combining static analysis and run-time checks. Systems like CCured [31] perform static analyses to check source code for safety, and automatically insert run-time checks where safety cannot be guaranteed statically.

These are good techniques — Cyclone itself uses the third strategy. However, except for CCured, none of the above projects applies them in a way that comes close to ruling out all of the safety violations found in C. It is not hard for a program to pass LINT and still crash, and even the more advanced checking systems, like LCLint, SLAM, and Metal, do not find all safety violations. We can say something similar about all of the other systems mentioned above. Furthermore, most of these systems are simply not used — `assert` is probably the most popular, but it is usually turned off when code is shipped to avoid performance degradation.

CCured and Cyclone both seek to rule out all safety violations. The main disadvantage of CCured is that it takes control away from programmers. CCured needs to maintain some extra bookkeeping information in order to perform necessary run-time checks, and it does this by modifying data representations. For example, an `int *` might be represented by just an address, but it might also be represented by an address plus extra data that allows bounds checking. This means that CCured has control over data representations, not the programmer; and, moreover, basic operations (dereferencing, pointer arithmetic) will have different costs,

depending on the decisions made by CCured. Furthermore, CCured relies on a garbage collector, so programmers have less control over memory management. All of these decisions were made because CCured is most concerned with porting legacy code with little or no change; Cyclone is concerned with preserving C's hallmark control over low-level details such as data representation and memory management, both when porting old code and writing new code.

## 7 Conclusion

Cyclone is a C dialect that prevents safety violations in programs using a combination of static analyses and inserted run-time checks. Cyclone's goal is to accommodate C's style of low-level programming, while providing the same level of safety guaranteed by high-level safe languages like Java — a level of safety that has not been achieved by previous approaches.

## References

- [1] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *ACM Conference on Programming Language Design and Implementation*, pages 290–301, June 1994.
- [2] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 203–213, June 2001.
- [3] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001. Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, May 2001.
- [4] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack-smashing attacks. In *USENIX Annual 2000 Technical Conference*, San Diego, California, June 2000.
- [5] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software, Practice, and Experience*, 30(7):775–802, 2000.
- [6] CERT. Denial-of-service attack via ping. Advisory CA-1996-26, December 18, 1996. <http://www.cert.org/advisories/CA-1996-26.html>.
- [7] CERT. Double free bug in zlib compression library. Advisory CA-2002-07, March 12, 2002. <http://www.cert.org/advisories/CA-2002-07.html>.
- [8] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *10th USENIX Security Symposium*, Washington, D.C., August 2001.
- [9] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, San Antonio, Texas, January 1998.
- [10] Cyclone. <http://www.cs.cornell.edu/projects/cyclone/>.
- [11] John DeVale and Philip Koopman. Performance evaluation of exception handling in I/O libraries. In *The International Conference on Dependable Systems and Networks*, June 2001.
- [12] Roman Drahtmueller. Re: SuSE Linux 6.x 7.0 Ident buffer overflow. Bugtraq mailing list, November 29, 2000. <http://www.securityfocus.com/archive/1/147592>.
- [13] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation*, October 2000.
- [14] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of Eighteenth ACM Symposium on Operating Systems Principles*, October 2001.
- [15] Chris Evans. “gdm” remote hole. Bugtraq mailing list, May 22, 2000. <http://www.securityfocus.com/archive/1/61099>.

- [16] Chris Evans. Very interesting traceroute flaw. Bugtraq mailing list, September 28, 2000. <http://www.securityfocus.com/archive/1/136215>.
- [17] David Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, May 1996.
- [18] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *4th USENIX Windows Systems Symposium*, August 2000.
- [19] Mike Frantzen and Mike Shuey. Stackghost: Hardware facilitated stack protection. In *10th USENIX Security Symposium*, Washington, D.C., August 2001.
- [20] Dan Grossman and Greg Morrisett. Scalable certification for typed assembly language. In *3rd International Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 117–145, Montreal, Canada, September 2000. Springer-Verlag.
- [21] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. ACM, June 2002.
- [22] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991.
- [23] Philip Koopman and John DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Transactions on Software Engineering*, 26(9), September 2000.
- [24] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, Washington, D.C., August 2001.
- [25] Elias Levy. Re: rpc.ttdbserverd on solaris 7. Bugtraq mailing list, November 19, 1999. <http://www.securityfocus.com/archive/1/35480>.
- [26] Greg McGary. Bounds checking projects. <http://www.gnu.org/software/gcc/projects/bp/main.html>.
- [27] MediaNet. <http://www.cs.cornell.edu/people/mhicks/medianet.htm>.
- [28] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of Unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.
- [29] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [30] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, 1999. Published as INRIA Technical Report 0288, March, 1999.
- [31] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Twenty-Ninth ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002. To appear.
- [32] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, Washington, D.C., August 2001.
- [33] Stephan Somogyi and Bruce Schneier. Inside risks: The perils of port 80. *Communications of the ACM*, 44(10), October 2001.
- [34] “tf8”. Wu-Ftpd remote format string stack overwrite vulnerability. Bugtraq vulnerability 1387, June 22, 2000. <http://www.securityfocus.com/bid/1387>.

# Cooperative Task Management without Manual Stack Management

or, Event-driven Programming is Not the Opposite of Threaded Programming

Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, John R. Douceur

Microsoft Research

1 Microsoft Way, Redmond, Washington 98052

{*adya, howell, theimer, bolosky, johndo*}@microsoft.com

## Abstract

Cooperative task management can provide program architects with ease of reasoning about concurrency issues. This property is often espoused by those who recommend “event-driven” programming over “multithreaded” programming. Those terms conflate several issues. In this paper, we clarify the issues, and show how one can get the best of both worlds: reason more simply about concurrency in the way “event-driven” advocates recommend, while preserving the readability and maintainability of code associated with “multithreaded” programming.

We identify the source of confusion about the two programming styles as a conflation of two concepts: *task management* and *stack management*. Those two concerns define a two-axis space in which “multithreaded” and “event-driven” programming are diagonally opposite; there is a third “sweet spot” in the space that combines the advantages of both programming styles. We point out pitfalls in both alternative forms of stack management, *manual* and *automatic*, and we supply techniques that mitigate the danger in the automatic case. Finally, we exhibit adaptors that enable automatic stack management code and manual stack management code to interoperate in the same code base.

## 1 Introduction

Our team embarked on a new project and faced the question of what programming model to use. Each team member had been burned by concurrency issues in the past, encountering bugs that were difficult to even reproduce, much less identify and remove. We chose to follow the collective wisdom of the community as we un-

derstood it, which suggests that an “event-driven” programming model can simplify concurrency issues by reducing opportunities for race conditions and deadlocks [Ous96]. However, as we gained experience, we realized that the popular term “event-driven” conflates several distinct concepts; most importantly, it suggests that a gain in reasoning about concurrency cannot be had without cumbersome manual stack management. By separating these concerns, we were able to realize the “best of both worlds.”

In Section 2, we define the two distinct concepts whose conflation is problematic, and we touch on three related concepts to avoid confusing them with the central ideas. The key concept is that one can choose the reasoning benefits of cooperative task management without sacrificing the readability and maintainability of automatic stack management. Section 3 focuses on the topic of stack management, describing how software evolution exacerbates problems both for code using manual stack management as well as code using automatic stack management. We show how the most insidious problem with automatic stack management can be alleviated. Section 4 presents our hybrid stack-management model that allows code using automatic stack management to co-exist and interoperate in the same program with code using manual stack management; this model helped us find peace within a group of developers that disagreed on which method to use. Section 5 discusses our experience in implementing these ideas in two different systems. Section 6 relates our observations to other work and Section 7 summarizes our conclusions.

## 2 Definitions

In this section, we define and describe five distinct concepts: *task management*, *stack management*, *I/O*



*response management, conflict management, and data partitioning*. These concepts are not completely orthogonal, but considering them independently helps us understand how they interact in a complete design. We tease apart these concerns and then return to look at how the concepts have been popularly conflated.

## 2.1 Task management

One can often divide the work a program does into conceptually separate tasks: each task encapsulates a control flow, and all of the tasks access some common, shared state. High-performance programs are often written with *preemptive* task management, wherein execution of tasks can interleave on uniprocessors or overlap on multiprocessors. The opposite approach, *serial* task management, runs each task to completion before starting the next task. Its advantage is that there is no conflict of access to the shared state; one can define inter-task invariants on the shared state and be assured that, while the present task is running, no other tasks can violate the invariants. The strategy is inappropriate, however, when one wishes to exploit multiprocessor parallelism, or when slow tasks must not defer later tasks for a long time.

A compromise approach is *cooperative* task management. In this approach, a task's code only yields control to other tasks at well-defined points in its execution; usually only when the task must wait for long-running I/O. The approach is valuable when tasks must interleave to avoid waiting on each other's I/O, but multiprocessor parallelism is not crucial for good application performance.

Cooperative task management preserves some of the advantage of serial task management in that invariants on the global state only need be restored when a task explicitly yields, and they can be assumed to be valid when the task resumes. Cooperative task management is harder than serial in that, if the task has local state that depends on the global state before yielding, that state may be invalid when the task resumes. The same problem appears in preemptive task management when releasing locks for the duration of a slow I/O operation [Bir89].

One penalty for adopting cooperative task management is that every I/O library function called must be wrapped so that instead of blocking, the function initiates the I/O and yields control to another task. The wrapper must also arrange for its task to become schedulable when the I/O completes.

## 2.2 Stack management

The common approach to achieving cooperative task management is to organize a program as a collection of event handlers. Say a task involves receiving a network message, reading a block from disk, and replying to the message. The receipt of the message is an event; one procedure handles that event and initiates the disk I/O. The receipt of the disk I/O result is a second event; another procedure handles that event and constructs the network reply message. The desired task management is achieved, in that other tasks may make progress while the present task is waiting on the disk I/O.

We call the approach just described *manual stack management*. As we argue in Section 3.1, the problem is that the control flow for a single conceptual task and its task-specific state are broken across several language procedures, effectively discarding language scoping features. This problem is subtle because it causes the most trouble as software evolves. It is important to observe that one can choose cooperative task management for its benefits while exploiting the *automatic* stack management afforded by a structured programming language. We describe how in Section 3.3.

Some languages have a built-in facility for transparently constructing closures; Scheme's call-with-current-continuation is an obvious example [HFW84, FHK84]. Such a facility obviates the idea of manual stack management altogether. This paper focuses on the stack management problem in conventional systems languages without elegant closures.

## 2.3 I/O management

While this paper focuses on the first two axes, we explicitly mention three other axes to avoid confusing them with the first two. The first concerns the question of *synchronous* versus *asynchronous I/O management*, which is orthogonal to the axis of task management. An I/O programming interface is *synchronous* if the calling task appears to block at the call site until the I/O completes, and then resume execution. An *asynchronous* interface call appears to return control to the caller immediately. The calling code may initiate several overlapping asynchronous operations, then later wait for the results to arrive, perhaps in arbitrary order. This form of concurrency is different than task management because I/O operations can be considered independently from the computation they overlap, since the I/O does not access the

shared state of the computation. Code obeying any of the forms of task management can call either type of I/O interface. Furthermore, with the right primitives, one can build wrappers to make synchronous interfaces out of asynchronous ones, and vice versa; we do just that in our systems.

## 2.4 Conflict management

Different task management approaches offer different granularities of atomicity on shared state. *Conflict management* considers how to convert available atomicity to a meaningful mechanism for avoiding resource conflicts. In serial task management, for example, an entire task is an atomic operation on shared state, so no explicit mechanism is needed to avoid inter-task conflicts on shared resources. In the limiting case of preemptive task management, where other tasks are executing concurrently, tasks must ensure that invariants hold on the shared state all the time.

The general solution to this problem is synchronization primitives, such as locks, semaphores, and monitors. Based on small atomic operations supplied by the machine or runtime environment, synchronization primitives let us construct mechanisms that maintain complex invariants on shared state that always hold. Synchronization mechanisms may be pessimistic or optimistic. A pessimistic mechanism locks other tasks out of the resources it needs to complete a computation. An optimistic primitive computes results speculatively; if the computation turns out to conflict with a concurrent task's computation, the mechanism retries, perhaps also falling back on a pessimistic mechanism if no forward progress is being made.

Cooperative (or serial) task management effectively provides arbitrarily large atomic operations: all of the code executed between two explicit yield points is executed atomically. Therefore, it is straightforward to build many complex invariants safely. This approach is analogous to the construction of atomic sequences with interrupt masking in uniprocessor OS kernels. We discuss in Section 3.3 how to ensure that code dependent on atomicity stays atomic as software evolves.

## 2.5 Data partitioning

Task management and conflict management work together to address the problem of potentially-concurrent

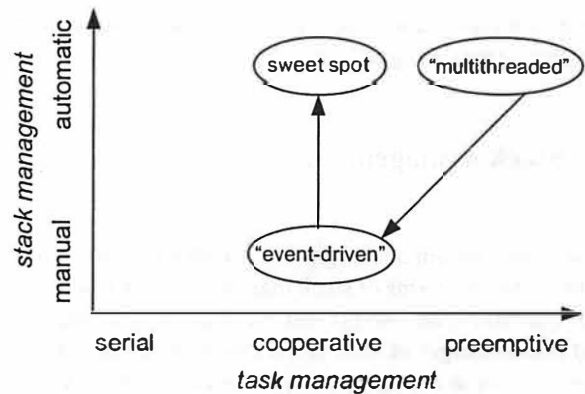


Figure 1: Two axes that are frequently conflated.

access to shared state. By partitioning that state, we can reduce the number of opportunities for conflict. For example, task-specific state needs no concurrency considerations because it has been explicitly *partitioned* from shared state. Data is transferred between partitions by value; one must be careful to handle implicit references (such as a value that actually indexes an array) thoughtfully.

Explicitly introducing data partitions to reduce the degree of sharing of shared state can make it easier to write and reason about invariants on each partition; data partitioning is an orthogonal approach to those mentioned previously.

## 2.6 How the concepts relate

We have described five distinct concepts. They are not all precisely orthogonal, but it is useful to consider the effects of choices in each dimension separately. Most importantly, for the purposes of this paper, the task management and stack management axes are indeed orthogonal (see Figure 1).

The idea behind Figure 1 is that conventional concurrent programming uses preemptive task management and exploits the automatic stack management of a standard language. We often hear this point in the space referred to by the term “threaded programming.” The second interesting point in the space is “event-driven programming,” where cooperative tasks are organized as event handlers that yield control by *returning* control to the event scheduler, manually unrolling their stacks. This paper is organized around the observation that one can choose cooperative task management while preserving the automatic stack management that makes a program-

ming language “structured,” in the diagram, this point is labeled the “sweet spot.”

### 3 Stack management

Given our diagram one might ask, “what are the *pros* and *cons* of the two forms of stack management? We address that question here. We present the principal advantages and disadvantages of each form, emphasizing how software evolution exacerbates the disadvantages of each. We also present a technique that mitigates the principal disadvantage of automatic stack management.

#### 3.1 Automatic versus manual

Programmers can express a task employing either *automatic* stack management or *manual* stack management. With automatic stack management, the programmer expresses each complete task as a single procedure in the source language. Such a procedure may call functions that block on I/O operations such as disk or remote requests. While the task is waiting on a blocking operation, its current state is kept in data stored on the procedure’s program stack. This style of control flow is one meaning often associated with the term “procedure-oriented.”

In contrast, manual stack management requires a programmer to rip the code for any given task into event handlers that run to completion without blocking. Event handlers are procedures that can be invoked by an *event-handling scheduler* in response to events, such as the initiation of a task or the response from a previously-requested I/O. To initiate an I/O, an event handler “ $E_1$ ” schedules a request for the operation but does not wait for the reply. Instead,  $E_1$  registers a task-specific object called a *continuation* [FHK84] with the event-handling scheduler. The continuation bundles state indicating where  $E_1$  left off working on the task, plus a reference to a different event-handler procedure  $E_2$  that encodes what should be done when the requested I/O has completed. After having initiated the I/O and registering the continuation,  $E_1$  returns control to the event-handling scheduler. When the event representing the I/O completion occurs, the event-handling scheduler calls  $E_2$ , passing  $E_1$ ’s bundled state as an argument. This style of control flow is often associated with the term “event-driven.”

To illustrate these two stack-management styles, con-

sider the code for a function, `GetCAInfo`, that looks in an in-memory hash table for a specified certificate-authority id and returns a pointer to the corresponding object. A certificate authority is an entity that issues certificates, for example for users of a file system.

```
CAInfo GetCAInfo(CAID caId) {
    CAInfo caInfo = LookupHashTable(caId);
    return caInfo;
}
```

Suppose that initially this function was designed to handle a few globally known certificate authorities and hence all the CA records could be stored in memory. We refer to such a function as a *compute-only* function: because it does not pause for I/O, we need not consider how its stack is managed across an I/O call, and thus the automatic stack management supplied by the compiler is always appropriate.

Now suppose the function evolves to support an abundance of CA objects. We may wish to convert the hash table into an on-disk structure, with an in-memory cache of the entries in use. `GetCAInfo` has become a function that may have to yield for I/O. How the code evolves depends on whether it uses automatic or manual stack management.

Following is code with automatic stack management that implements the revised function:

```
CAInfo GetCAInfoBlocking(CAID caId) {
    CAInfo caInfo = LookupHashTable(caId);
    if (caInfo != NULL) {
        // Found node in the hash table
        return caInfo;
    }
    caInfo = new CAInfo();
    // DiskRead blocks waiting for
    // the disk I/O to complete.
    DiskRead(caId, caInfo);
    InsertHashTable(caId, caInfo);
    return caInfo;
}
```

To achieve the same goal using manual stack management, we rip the single conceptual function `GetCAInfoBlocking` into two source-language functions, so that the second function can be called from the event-handler scheduler to continue after the disk I/O has completed. Here is the continuation object that stores the bundled state and function pointer:

```
class Continuation {
```

```

// The function called when this
// continuation is scheduled to run.
void (*function)(Continuation cont);
// Return value set by the I/O operation.
// To be passed to continuation.
void *returnValue
// Bundled up state
void *arg1, *arg2, ...;
}

```

Here is the original function, ripped into the two parts that function as event handlers:

```

void GetCAInfoHandler1(CAID caId,
                      Continuation *callerCont)
{
    // Return the result immediately if in cache
    CAInfo *caInfo = LookupHashTable(caId);
    if (caInfo != NULL) {
        // Call caller's continuation with result
        (*callerCont->function)(caInfo);
        return;
    }

    // Make buffer space for disk read
    caInfo = new CAInfo();
    // Save return address & live variables
    Continuation *cont = new
        Continuation(&GetCAInfoHandler2,
                    caId, caInfo, callerCont);
    // Send request
    EventHandle eh =
        InitAsyncDiskRead(caId, caInfo);
    // Schedule event handler to run on reply
    // by registering continuation
    RegisterContinuation(eh, cont);
}

void GetCAInfoHandler2(Continuation
*cont) {
    // Recover live variables
    CAID caId = (CAID) cont->arg1;
    CAInfo *caInfo = (CAInfo*) cont->arg2;
    Continuation *callerCont =
        (Continuation*) cont->arg3;
    // Stash CAInfo object in hash
    InsertHashTable(caId, caInfo);
    // Now "return" results to original caller
    (*callerCont->function)(callerCont);
}

```

Note that the signature of `GetCAInfo` is different from that of `GetCAInfoHandler1`. Since the desired re-

sult from what used to be `GetCAInfo` will not be available until `GetCAInfoHandler2` runs sometime later, the caller of `GetCAInfoHandler1` must pass in a continuation that `GetCAInfoHandler2` can later invoke in order to return the desired result via the continuation record. That is, with manual stack management, a statement that returns control (and perhaps a value) to a caller must be simulated by a function call to a continuation procedure.

## 3.2 Stack Ripping

In conventional systems languages, such as C++, which have no support for closures, the programmer has to do a substantial amount of manual stack management to yield for I/O operations. Note that the function in the previous section was ripped into two parts because of one I/O call. If there are more I/O calls, there are even more rips in the code. The situation gets worse still with the presence of control structures such as *for* loops. The programmer deconstructs the language stack, reconstructs it on the heap, and reduces the readability of the code in the process.

Furthermore, debugging is impaired because when the debugger stops in `GetCAInfoHandler2`, the call stack only shows the state of the current event handler and provides no information about the sequence of events that the ripped task performed before arriving at the current event handler invocation. Theoretically, one can manually recover the call stack by tracing through the continuation objects; in practice we have observed that programmers hand-optimize away tail calls, so that much of the stack goes missing.

In summary, for each routine that is ripped, the programmer will have to manually manage procedural language features that are normally handled by a compiler:

**function scoping** Now two or more language functions represent a single conceptual function.

**automatic variables** Variables once allocated on the stack by the language must be moved into a new state structure stored on the heap to survive across yield points.

**control structures** The entry point to every basic block containing a function that might block must be reachable from a continuation, and hence must be a separate language-level function. That is, conceptual functions with loops must be ripped into more than two pieces.

**debugging stack** The call stack must be manually recovered when debugging, and manual optimization of tail calls may make it unrecoverable.

Software evolution substantially magnifies the problem of function ripping: when a function evolves from being compute-only to potentially yielding, *all* functions, along every path from the function whose concurrency semantics have changed to the root of the call graph may potentially have to be ripped in two. (More precisely, all functions up a branch of the call graph will have to be ripped until a function is encountered that already makes its call in continuation-passing form.) We call this phenomenon “stack ripping” and see it as the primary drawback to manual stack management. Note that, as with all global evolutions, functions on the call graph may be maintained by different parties, making the change difficult.

### 3.3 Hidden concurrency assumptions

The huge advantage of manual stack management is that every yield point is explicitly visible in the code at every level of the call graph. In contrast, the call to `DiskRead` in `GetCAInfo` hides potential concurrency. Local state extracted from shared state before the `DiskRead` call may need to be reevaluated after the call. Absent a comment, the programmer cannot tell which function calls may yield and which local state to revalidate as a consequence thereof.

As with manual stack management, software evolution makes the situation even worse. A call that did not yield yesterday may be changed tomorrow to yield for I/O. However, when a function with manual stack management evolves to yield for I/O, its signature changes to reflect the new structure, and the compiler will call attention to any callers of the function unaware of the evolution. With automatic stack management, such a change is syntactically invisible and yet it affects the semantics of *every* function that calls the evolved function, either directly or transitively.

The dangerous aspect of automatic stack management is that a semantic property (*yielding*) of a called procedure dramatically affects how the calling procedure should be written, but there is no check that the calling procedure is honoring the property. Happily, concurrency assumptions can be declared explicitly and checked statically or dynamically.

A static check would be ideal because it detects viola-

tions at compile time. Functions that yield are tagged with the *yielding* property, and each block of a calling function that assumes that it runs without yielding is marked *atomic*. The compiler or a static tool checks that functions that call *yielding* functions are themselves marked *yielding*, and that no calls to *yielding* functions appear inside *atomic* blocks. In fact, one could reasonably abuse an exception declaration mechanism to achieve this end.

A dynamic check is less desirable than a static one because violations are only found if they occur at runtime. It is still useful in that violations cause an immediate failure, rather than subtly corrupting system state in a way that is difficult to trace back to its cause. We chose a dynamic check because it was quick and easy to implement. Each block of code that depends on atomicity begins with a call to `startAtomic()` and ends with a call to `endAtomic()`. The `startAtomic()` function increments a private counter and `endAtomic()` decrements it. When any function tries to block on I/O, `yield()` asserts that the counter is zero, and dumps core otherwise.

Note that in evolving code employing automatic stack management, we may also have to modify every function extending along every path up the call graph from a function whose concurrency semantics have changed. However, whereas manual stack management implies that each affected function must be torn apart into multiple pieces, automatic-stack-management code may require no changes or far less intrusive changes. If the local state of a function does not depend on the yielding behavior of a called function, then the calling function requires no change. If the calling function's local state is affected, the function must be modified to revalidate its state; this surgery is usually local and does not require substantial code restructuring.

## 4 Hybrid approach

In our project there are passionate advocates for each of the two styles of stack management. There is a hybrid approach that enables both styles to coexist in the same code base, using adaptors to connect between them. This hybrid approach also enables a project to be written in one style but incorporate legacy code written in the other.

In the Windows operating system, “threads” are scheduled preemptively and “fibers” are scheduled cooperatively. Our implementation achieves cooperative task management by scheduling multiple fibers on a single



thread; at any given time, only one fiber is active.

In our design, a scheduler runs on a special fiber called `MainFiber` and schedules both manual stack management code (event handlers) and automatic stack management code. Code written with automatic stack management, that expects to block for I/O, always runs on a fiber other than `MainFiber`; when it blocks, it always yields control back to `MainFiber`, where the scheduler selects the next task to schedule. Compute-only functions, of course, may run on any fiber, since they may be freely called from either context.

Both types of stack management code are scheduled by the same scheduler because the Windows fiber package only supports the notion of explicitly switching from one fiber to another specified fiber; there is no notion of a generalized yield operation that invokes a default fiber scheduler. Implementing a combined scheduler also allowed us to avoid the problem of having two, potentially conflicting, schedulers running in parallel: one for event handlers and one for fibers.

There are other ways in which the two styles of code can be made to interact. We aimed for simplicity and to preserve our existing code base that uses manual stack management. Our solution ensures that code written in either style can call a function implemented in the other style without being aware that the other stack management discipline even exists.

To illustrate the hybrid approach, we show an example that includes calls across styles in both directions. The example involves four functions: `FetchCert`, `GetCertData`, `VerifyCert`, and `GetCAInfo`. (`GetCAInfo` was introduced in Section 3.1). `FetchCert` fetches a security certificate using `GetCertData` and then calls `VerifyCert` in order to confirm its validity. `VerifyCert`, in turn, calls `GetCAInfo` in order to obtain a CA with which to verify a certificate. Here is how the code would look with serial task management:

```
bool FetchCert(User user,
               Certificate *cert) {
    // Get the certificate data from a
    // function that might do I/O
    certificate = GetCertData(user);
    if (!VerifyCert(user, cert)) {
        return false;
    }
}

bool VerifyCert(User user,
```

```
        Certificate *cert) {
    // Get the Certificate Authority (CA)
    // information and then verify cert
    ca = GetCAInfo(cert);
    if (ca == NULL) return false;
    return CACheckCert(ca, user, cert);
}
```

```
Certificate* GetCertData(User user) {
    // Look up certificate in the memory
    // cache and return the answer.
    // Else fetch from disk/network
    if (Lookup(user, cert))
        return certificate;
    certificate = DoIOAndGetCert();
    return certificate;
}
```

Of course, we want to rewrite the code to use cooperative task management, allowing other tasks to run during the I/O pauses, with different functions adhering to each form of stack management. Suppose that `VerifyCert` is written with automatic stack management and the remaining functions (`FetchCert`, `GetCertData`, `GetCAInfo`) are implemented with manual stack management (using continuations). We will define adaptor functions that route control flow between the styles.

## 4.1 Manual calling automatic

Figure 2 is a sequence diagram illustrating how code with manual stack management calls code with automatic stack management. In the figure, the details of a call in the opposite direction are momentarily obscured behind dashed boxes. The first event handler for `FetchCert1` calls the function `GetCertData1`, which initiates an I/O operation, and the entire stack unrolls in accordance with manual stack management. Later, when the I/O reply arrives, the scheduler executes the `GetCertData2` continuation, which “returns” (by a function call) to the second handler for `FetchCert`. This is pure manual stack management.

When a function written with manual stack management calls code with automatic stack management, we must reconcile the two styles. The caller code is written expecting never to block on I/O; the callee expects to block I/O always. To reconcile these styles, we create a new fiber and execute the callee code on that fiber. The caller resumes (to manually unroll its stack) as soon as the first burst of execution on the fiber completes. The fiber may



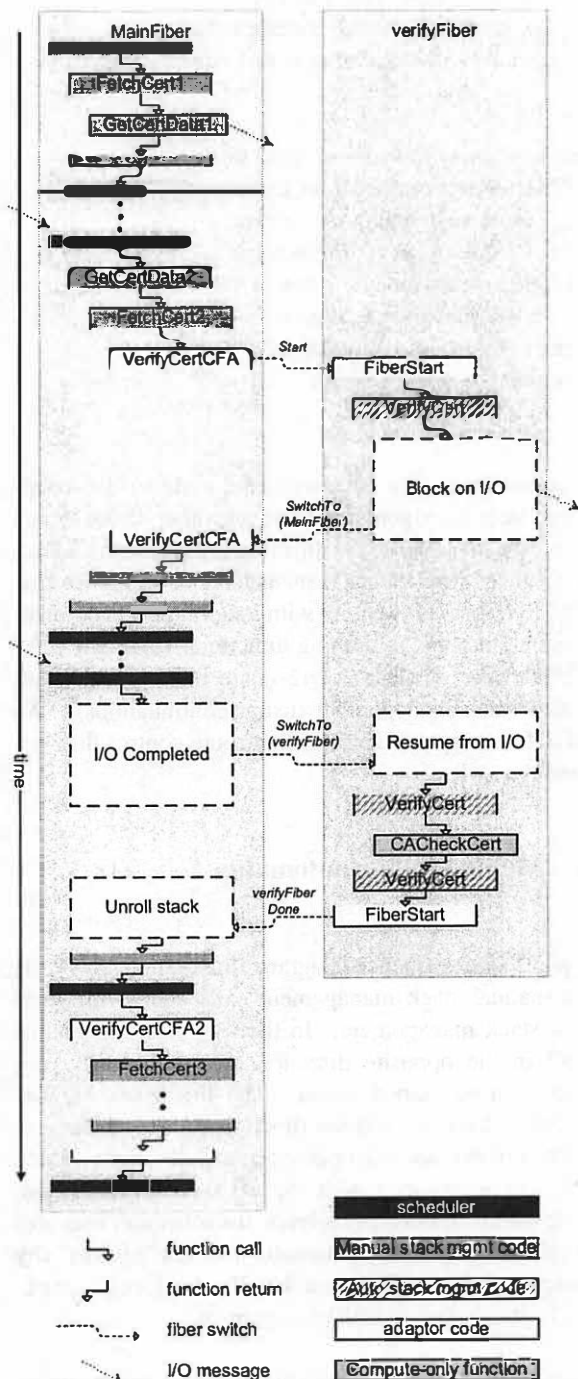


Figure 2: GetCertData, code with manual stack management, calls VerifyCert, a function written with automatic stack management.

run and block for I/O several times; when it finishes its work on behalf of the caller, it executes the caller's continuation to resume the caller's part of the task. Thus, the caller code does not block and the callee code can block if it wishes.

In our example, the manual-stack-management function FetchCert2 calls through an adaptor to the automatic-stack-management function VerifyCert. FetchCert2 passes along a continuation pointing at FetchCert3 so that it can eventually regain control and execute the final part of its implementation. The following code is for the CFA adaptor, ripped into its *call* and *return* parts; CFA stands for "Continuation-To-Fiber adaptor."

```
void VerifyCertCFA(CertData certData,
                  Continuation *callerCont) {
    // Executed on MainFiber
    Continuation *vcaCont = new
        Continuation(VerifyCertCFA2,
                    callerCont);

    Fiber *verifyFiber = new
        VerifyCertFiber(certData, vcaCont);
    // On fiber verifyFiber, start executing
    // VerifyCertFiber::FiberStart
    SwitchToFiber(verifyFiber);
    // Control returns here when
    // verifyFiber blocks on I/O
}
```

```
void VerifyCertCFA2(Continuation
*vcaCont) {
    // Executed on MainFiber.
    // Scheduled after verifyFiber is done
    Continuation *callerCont =
        (Continuation*) vcaCont->arg1;
    callerCont->returnValue =
        vcaCont->returnValue;
    // "return" to original caller (FetchCert)
    (*callerCont->function)(callerCont);
}
```

The first adaptor function accepts the arguments of the adapted function and a continuation ("stack frame") for the calling task. It constructs its own continuation vcaCont and creates a object called verifyFiber that represents a new fiber (VerifyCertFiber is a subclass of the Fiber class); this object keeps track of the function arguments and vcaCont so that it can transfer control to VerifyCertCFA2 when verifyFiber's work is done. Finally, it performs a fiber-switch to verifyFiber. When verifyFiber begins, it executes glue routine

VerifyCertFiber::FiberStart to unpack the parameters and pass them to VerifyCert, which may block on I/O:

```
VerifyCertFiber::FiberStart() {
    // Executed on a fiber other than MainFiber
    // The following call could block on I/O.
    // Do the actual verification.
    this->vcaCont->returnValue =
        VerifyCert(this->certData);
    // The verification is complete.
    // Schedule VerifyCertCFA2
    scheduler->schedule(this->vcaCont);
    SwitchTo(MainFiber);
}
```

This start function simply calls into the function VerifyCert. At some point, when VerifyCert yields for I/O, it switches control back to the MainFiber using a SwitchTo call in the I/O function (not the call site shown in the FiberStart() routine above). Control resumes in VerifyCertCFA, which unrolls the continuation stack (i.e., GetCertData2 and FetchCert2) back to the scheduler. Thus, the hybrid task has blocked for the I/O initiated by the code with automatic stack management while ensuring that event handler FetchCert2 does not block.

Later, when the I/O completes, verifyFiber is resumed (for now, we defer the details on how this resumption occurs). After VerifyCert has performed the last of its work, control returns to FiberStart. FiberStart stuffs the return value into VerifyCertCFA2's continuation, schedules it to execute, and switches back to the MainFiber a final time. At this point, verifyFiber is destroyed. When VerifyCertCFA2 executes, it "returns" (with a function call, as code with manual stack management normally does) the return value from VerifyCert back to the adaptor-caller's continuation, FetchCert3.

## 4.2 Automatic calling manual

We now discuss how the code interactions occur when a function with automatic stack management calls a function that manually manages its stack. In this case, the former function needs to block for I/O, but the latter function simply schedules the I/O and returns. To reconcile these requirements, we supply an adaptor that calls the manual-stack-management code with a special continuation and relinquishes control to the MainFiber,

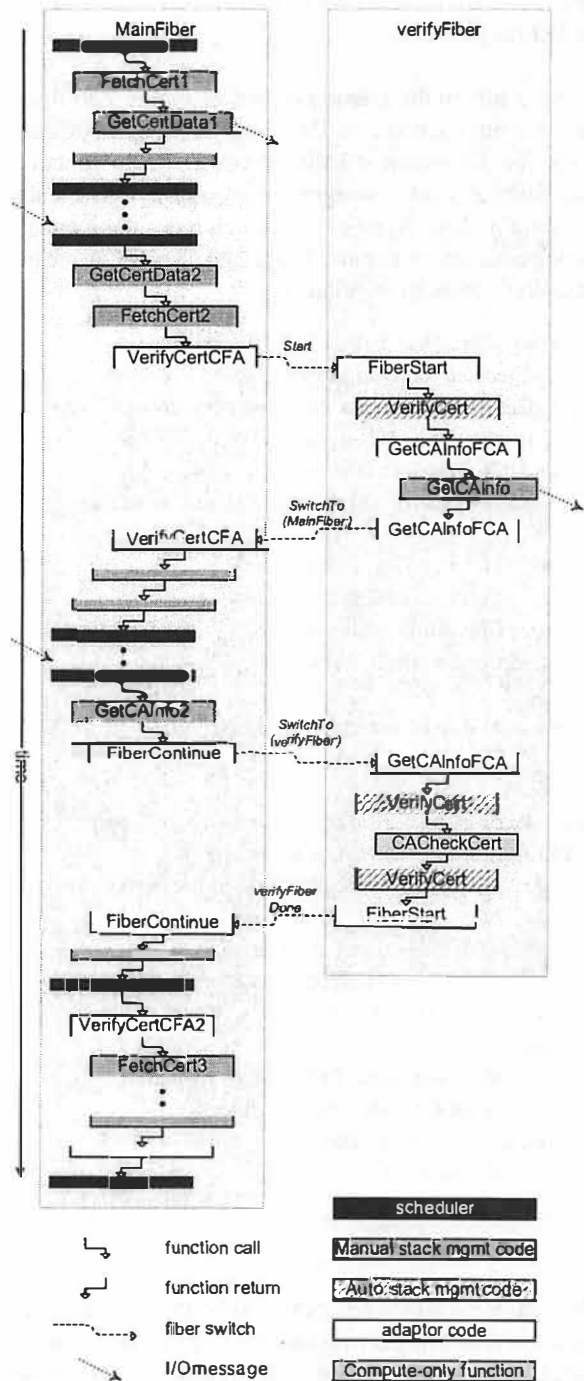


Figure 3: VerifyCert, code with automatic stack management, calls GetCAInfo, a function written with manual stack management.

thereby causing the adaptor's caller to remain blocked. When the I/O completes, the special continuation runs on the MainFiber and resumes the fiber of the blocked adaptor, which resumes the original function waiting for the I/O result.

Figure 3 fills in the missing details of Figure 2 to illustrate this interaction. In this example, VerifyCert blocks on I/O when it calls GetCAInfo, a function with manual stack management. VerifyCert calls the adaptor GetCAInfoFCA, which hides the manual-stack-management nature of GetCAInfo (FCA means Fiber-to-Continuation Adaptor):

```
Boolean GetCAInfoFCA(CAID caid) {
    // Executed on verifyFiber
    // Get a continuation that switches control
    // to this fiber when called on MainFiber
    FiberContinuation *cont = new
        FiberContinuation(FiberContinue,
                          this);
    GetCAInfo(caid, cont);
    if (!cont->shortCircuit) {
        // GetCAInfo did block.
        SwitchTo(MainFiber);
    }
    return cont->returnValue;
}

void FiberContinue(Continuation *cont) {
    if (!Fiber::OnMainFiber()) {
        // Manual stack mgmt code did not perform
        // I/O: just mark it as short-circuited
        FiberContinuation *fcont =
            (FiberContinuation) *cont;
        fcont->shortCircuit = true;
    } else {
        // Resumed after I/O: simply switch
        // control to the original fiber
        Fiber *f = (Fiber *) cont->arg1;
        f->Resume();
    }
}
```

The adaptor, GetCAInfoFCA, sets up a special continuation that will later resume verifyFiber via the code in FiberContinue. It then passes this continuation to GetCAInfo which initiates an I/O operation and returns immediately to what it believes to be the event-handling scheduler; of course, in this case, the control returns to GetCAInfoFCA. Since I/O was scheduled and short-circuiting did not occur (discussed later in this section), GetCAInfoFCA must ensure that control does not yet return to VerifyCert; to achieve this

effect, it switches control to the MainFiber.

On the MainFiber, the continuation code that started this burst of fiber execution, VerifyCertCFA, returns several times to unroll its stack and the scheduler runs again. Eventually, the I/O result arrives and the scheduler executes GetCAInfo2, the remaining work of GetCAInfo. GetCAInfo2 fills the local hash table (recall its implementation from Section 3.1) and "returns" control by calling a continuation. In this case, it calls the continuation (FiberContinue) that had been passed to GetCAInfo.

FiberContinue notices that verifyFiber has indeed been blocked and switches control back to that fiber, where the bottom half of the adaptor, GetCAInfoFCA, extracts the return value and passes it up to the automatic-stack-management code that called it (VerifyCert).

The *short circuit* branch not followed in the example handles the case where GetCAInfo returns a result immediately without waiting for I/O. When it can do so, it must *not* allow control to pass to the scheduler. This is necessary so that a caller can optionally determine whether or not a routine has yielded control and hence whether or not local state must be revalidated. Without a *short circuit* path, this important optimization and an associated design pattern that we describe in Section 5 cannot be achieved. Figure 4 illustrates the short-circuit sequence: The short-circuit code detects the case where GetCAInfo runs locally, performs no I/O, and executes ("returns to") the current continuation immediately. FiberContinue detects that it was not executed directly by the scheduler, and sets the shortCircuit flag to prevent the adaptor from switching to the MainFiber.

### 4.3 Discussion

An important observation is that, with adaptors in place, each style of code is unaware of the other. A function written with automatic stack management sees what it expects: deep in its stack, control may transfer away, and return later with the stack intact. Likewise, the event-handler scheduler cannot tell that it is calling anything other than just a series of ordinary manual-stack-management continuations: the adaptors deftly swap the fiber stacks around while looking like any other continuation. Thus, integrating code in the two styles is straightforward: fiber execution looks like a continuation to the event-driven code, and the continuation scheduler looks

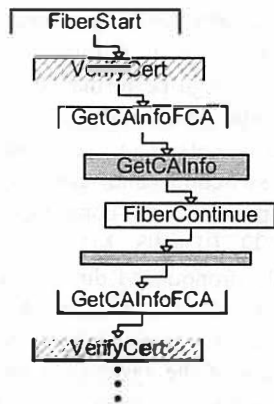


Figure 4: A variation where GetCAInfo does not need to perform I/O.

like any other fiber to the procedure-oriented code. This adaptability enables automatic-stack-management programmers to work with manual-stack-management programmers, and to evolve a manual-stack-management code base with automatic-stack-management functions and *vice versa*.

## 5 Implementation Experience

We have employed cooperative task management in two systems: Farsite [BDET00], a distributed, secure, serverless file system running over desktops, and UCoM [SBS02], a wireless phone application for handheld devices such as PDAs. The Farsite code is designed to run as a daemon process servicing file requests on the Windows NT operating system. The UCoM system, designed for the Windows CE operating system, is a client application that runs with UI and audio support.

The Farsite system code was initially written in event-driven style (cooperative task management and manual stack management) to enable simplified reasoning about the concurrency conditions of the system. As our code base grew and evolved over a period of two years, we came to appreciate the costs of employing manual stack management and devised the hybrid approach discussed in the previous section to introduce automatic stack management code into our system. The UCoM system uses automatic stack management exclusively.

Farsite uses fibers, the cooperative threading facility available in Windows NT. With Windows fibers, each task's state is represented with a stack, and control is transferred by simply swapping one stack pointer for

another, as with `setjmp` and `longjmp`. Since fibers are unavailable in the Windows CE operating system, UCoM uses preemptive threads and condition variables to achieve a cooperative threading facility: each thread blocks on its condition variable and the scheduler ensures that at most one condition variable is signalled at any moment. When a thread yields, it blocks on its condition variable and signals the scheduler to continue; the scheduler selects a ready thread and signals its condition variable.

We implemented the hybrid adaptors in each direction with a series of mechanically-generated macros. There are two groups of macros, one for each direction of adaptation. Within each group, there are variations to account for varying numbers of arguments, void or non-void return type, and whether the function being called is a static function or an object method; multiple macros are necessary to generate the corresponding variations in syntax. Each macro takes as arguments the signature of the function being adapted. The macros declare and create appropriate Fiber and Continuation objects.

Our experience with both systems has been positive and our subjective impression is that we have been able to preempt many subtle concurrency problems by using cooperative task management as the basis for our work. Although the task of wrapping I/O functions (see Section 2.1) can be tedious, it can be automated, and we found that paying an up-front cost to reduce subtle race conditions was a good investment.

Both systems use extra threads for converting blocking I/O operations to non-blocking operations and for scheduling I/O operations, as is done in many other systems, such as Flash [PDZ99]. Data partitioning prevents synchronization problems between the I/O threads and the state shared by cooperatively-managed tasks.

Cooperative task management avoids the concurrency problems of locks only if tasks can complete without having to yield control to any other task. To deal with tasks that need to perform I/O, we found that we could often avoid the need for a lock by employing a particular design pattern. In this pattern, which we call the *Pinning Pattern*, I/O operations are used to *pin* resources in memory where they can be manipulated without yielding. Note that *pinning* does not connote exclusivity: a *pinned* resource is held in memory (to avoid the need to block on I/O to access it), but when other tasks run, they are free to manipulate the data structures it contains. Functions are structured in two phases: a loop that repeatedly tries to execute all potentially-yielding operations until they can all be completed without yielding,

and an atomic block that computes results and writes them into the shared state.

An important detail of the design pattern is that there may be dependencies among the potentially-yielding operations. A function may need to compute on the results of a previously-pinned resource in order to decide which resource to pin next; for example, in Farsite this occurs when traversing a path in a directory tree. Thus, in the fully general version of the design pattern, a check after each potentially-yielding operation ascertains whether the operation did indeed yield, and if so, restarts the loop from the top. Once the entire loop has executed without interruption, we know that the set of resources we have pinned in memory are related in the way we expect, because the final pass through the loop executed atomically.

## 6 Related Work

Birrell offers a good overview of the conventional thread-ed programming model with preemptive task management [Bir89]. Of his reasons for using concurrency (p. 2), cooperative task management can help with all but exploiting multiprocessors, a shortcoming we mention in Section 2.1. Birrell advises that “you must be fastidious about associating each piece of data with one (and only one) mutex” (p. 28); consider cooperative task management as the limiting case of that advice. There is the complexity that whenever a task yields it effectively releases the global mutex, and must reestablish its invariants when it resumes. But even under preemptive task management, Birrell comments that “you might want to unlock the mutex before calling down to a lower level abstraction that will block or execute for a long time” (p. 12); hence this complexity is not introduced by the choice of cooperative task management.

Ousterhout points out the pitfalls of preemptive task management, such as subtle race conditions and deadlocks [Ous96]. We argue that his “threaded” model conflates preemptive task management with automatic stack management, and his “event-driven” model conflates cooperative task management with manual stack management. We wish to convince designers that the choices are orthogonal, that Ousterhout’s arguments are really about the task management decision, and that programmers should exploit the ease-of-reasoning benefits of cooperative task management while exploiting the features of their programming language by using automatic stack management.

Other system designers have advocated non-threaded programming models because they observe that for a certain class of high-performance systems, such as file servers and web servers, substantial performance improvements can be obtained by reducing context switching and carefully implementing application-specific cache-conscious task scheduling [HS99, PDZ99, BDM98, MY98]. These factors become especially pronounced during high load situations, when the number of threads may become so large that the system starts to thrash while trying to give each thread its fair share of the system’s resources. We argue that the context-switching overhead for user-level threads (fibers) is in fact quite low; we measured the cost of switching in our fiber package to be less than ten times the cost of a procedure call. Furthermore, application-specific cache-conscious task scheduling should be just as achievable with cooperative task management and automatic stack management: the scheduler is given precisely the same opportunities to schedule as in event-driven code; the only difference is whether stack state is kept on stacks or in chains of continuations on the heap.

For the classes of applications we reference here, processing is often partitioned into stages [WCB01, LP01]. The partitioning of system state into disjoint stages is a form of data partitioning, which addresses concurrency at the coarse grain. Within each stage, the designer of such a system must still choose a form of conflict management, task management, and stack management. Careful construction of stages avoids I/O calls within a stage; in that case, cooperative task management within the stage degenerates to serial task management, and no distinction arises in stack management. In practice, at the inter-stage level, a single task strings through multiple stages, and reads as in manual stack management. Typically, the stages are monotonic: once a task leaves a stage, it never returns. This at least avoids the ripping associated with looping control structures.

Lauer and Needham show two programming models to be equivalent up to syntactic substitution [LN79]. We describe their models in terms of our axes: their procedure-oriented system has preemptive task management, automatic stack management (“a process typically has only one goal or task”), monitors for conflict management, and one big data partition protected by those monitors. Their message-oriented system has manual stack management with task state passed around in messages, and no conflicts to manage due to many partitions of the state so that it is effectively never concurrently shared.

Notably, of the message-oriented system, they say “nei-



ther procedural interfaces nor global naming schemes are very useful,” that is, the manual stack management undermines structural features of the language. Neither model uses cooperative task management as we regard it, since both models require identically-detailed reasoning about conflict management. Thus their comparison is decidedly not between the models we associate with *multithreaded* and *event-driven* programming.

## 7 Conclusions

In this paper we clarify an ongoing debate about “event-driven” versus “threaded” programming models by identifying two separable concerns: task management and stack management. Thus separated, the paper assumes cooperative task management and focuses on issues of stack management in that context. Whereas the choice of task management strategy is fundamental, the choice of stack management can be left to individual taste. Unfortunately, the term “event-driven programming” conflates both cooperative task management and manual stack management. This prevents many people from considering using a readable automatic-stack-management coding style in conjunction with cooperative task management.

Software evolution is an important factor affecting the choice of task management strategy. When concurrency assumptions evolve it may be necessary to make global, abstraction-breaking changes to an application’s implementation. Evolving code with manual stack management imposes the cumbersome code restructuring burden of stack ripping; evolving either style of code involves revisiting the invariant logic due to changing concurrency assumptions and sometimes making localized changes to functions in order to revalidate local state.

Finally, a hybrid model adapts between code with automatic and with manual stack management, enabling cooperation among disparate programmers and software evolution of disparate code bases.

## 8 Acknowledgements

We would like to thank Jim Larus for discussing this topic with us at length. Thanks also to the anonymous reviewers for their thoughtful comments.

## References

- [BDET00] William Bolosky, John Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proceedings of the ACM Sigmetrics 2000 Conference*, pages 34–43, June 2000.
- [BDM98] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Better operating system features for faster network servers. In *Proceedings of the Workshop on Internet Server Performance (held in conjunction with ACM SIGMETRICS ’98)*, Madison, WI, 1998.
- [Bir89] Andrew D. Birrell. An introduction to programming with threads. Technical Report 35, Digital Systems Research Center, January 1989.
- [FHK84] Daniel P. Friedman, Christopher T. Haynes, and Eugene E. Kohlbecker. Programming with continuations. In P. Pepper, editor, *Program Transformation and Programming Environments*, pages 263–274. Springer-Verlag, 1984.
- [HFW84] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *ACM Symposium on LISP and Functional Programming*, pages 293–298, Austin, TX, August 1984.
- [HS99] J. Hu and D. Schmidt. JAWS: A Framework for High Performance Web Servers. In *Domain-Specific Application Frameworks: Frameworks Experience by Industry*. Wiley & Sons, 1999.
- [LN79] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, January 1979.
- [LP01] J. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. Technical Report MSR-TR-2001-39, Microsoft Research, March 2001.
- [MY98] S. Mishra and R. Yang. Thread-based vs event-based implementation of a group communication service. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, 1998.



- [Ous96] John Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX Technical Conference (Invited Talk)*, Austin, TX, January 1996.
- [PDZ99] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Technical Conference*, Monterey, CA, June 1999.
- [SBS02] E. Shih, P. Bahl, and M. Sinclair. Wake on Wireless: An event driven power saving strategy for battery operated devices. Technical Report MSR-TR-2002-40, Microsoft Research, April 2002.
- [WCB01] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October 2001.

# Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems \*

Hai Huang, Padmanabhan Pillai, and Kang G. Shin

*Real-Time Computing Laboratory*

*Department of Electrical Engineering and Computer Science*

*The University of Michigan*

*Ann Arbor, MI 48109-2122*

*{haih,pillai,kgshin}@eecs.umich.edu*

## Abstract

Concurrency management is a basic requirement for interprocess communication in any multitasking system. This usually takes the form of lock-based or other blocking algorithms. In real-time and/or time-sensitive systems, the less-predictable timing behavior of lock-based mechanisms and the additional task-execution dependency make synchronization undesirable. Recent research has provided non-blocking and wait-free algorithms for interprocess communication, particularly in the domain of single-writer, multiple-reader semantics, but these algorithms typically incur high costs in terms of computation or space complexity, or both. In this paper, we propose a general transformation mechanism that takes advantage of temporal characteristics of the system to reduce both time and space overheads of current single-writer, multiple-reader algorithms. We show a 17–66% execution time reduction along with a 14–70% memory space reduction when three wait-free algorithms are improved by applying our transformation. We present three new algorithms for wait-free, single-writer, multiple-reader communication along with detailed performance evaluation of nine algorithms under various experimental conditions.

## 1 Introduction

A key benefit provided by operating systems is a task or thread abstraction to manage the complexity that rapidly evolves even in very small embedded systems. A task/thread model mitigates the complexity growth of large monolithic programs, and simplifies the sharing of computing resources between the disparate functions of the system. However, the tasks of a system very rarely work independently of each other, hence needing interprocess communication (IPC) between tasks.

The simplest method of IPC is through global, shared variables. This is a very low-overhead method of communication,

but has obvious flaws in concurrent accesses by multiple tasks. Even if we restrict the domain to single-writer semantics, which is common in embedded systems and sensor networks, data corruption can occur.

To avoid reading corrupted data from a concurrent object, critical sections are often used to coordinate accesses from different tasks. The simplest approach to implementing critical sections is to disallow task preemption inside of the critical section. This can be done by disabling and enabling interrupts in the CPU at the beginning and end of the critical sections, respectively. These are privileged operations and require kernel intervention. The read and write operations must be implemented in the kernel, or the application must be wholly trusted, since any task running with interrupts disabled cannot be preempted and may, either maliciously or inadvertently, disrupt the system. Moreover, disabling interrupts does not suffice to manage concurrency in multiprocessor systems.

The most common way to implement critical sections is to use software locks — typically through mutexes and semaphores. A task has to acquire the necessary locks before it can access shared objects. If the needed lock is already held by another task, the task blocks, and the operating system will resume it when the resource becomes available. Using locks serializes concurrent tasks that try to access the shared objects simultaneously, thus preventing corruption. In a multiprocessor environment, this reduces parallelism and decreases the utilization of available resources.

Locks can also cause more serious problems such as unpredictable blocking times and deadlocks. If a task is blocked while still holding the lock (e.g., a page fault occurred, or it is preempted by a higher-priority task), any other tasks waiting for the lock are unable to make progress until the lock is subsequently released. In the worst case, the task may fail while holding the lock, or block indefinitely due to circular lock dependencies, causing deadlock and blocking other tasks from ever making progress.

Even with safeguards to avoid deadlock, locks are particu-

\*The work reported in this paper is supported in part by the U.S. Airforce Office of Scientific Research under Grant No. F49620-01-1-0120, and by DARPA administered under AFRL contract F30602-01-02-0527.

larly unattractive in real-time and embedded systems. Due to blocking and switching to other tasks, using locks can incur high and unpredictable execution time overheads, and cause many other problems, including priority inversion, convoying of tasks, more difficult schedulability analysis, and increased susceptibility to faults. In real-time systems, tasks are usually assigned fixed or deadline-based priorities, according to which they are scheduled. Priority inversion can occur when a high-priority task is blocked waiting for a lock, but the lock holder does not make progress due to its low priority. This is such a serious issue that many algorithms have been developed to limit the effects of priority inversion, including the priority inheritance protocol, the priority ceiling protocol, and the immediate priority ceiling protocol [3, 28, 29]. Furthermore, providing real-time execution guarantees becomes more difficult. The simple, classical real-time analysis techniques [21] assume independently-executing tasks, which is clearly violated when locks are used. More complex analysis [29] may be used to provide real-time guarantees by accounting for worst-case blocking times, but this may result in poorer utilization of system resources.

Due to the above problems associated with lock-based synchronization IPC approaches, several algorithms that perform non-blocking and wait-free<sup>1</sup> communication with single-writer, multiple-reader semantics have been proposed. These allow tasks to independently access the shared message area without locks and the problems introduced by blocking. These algorithms, however, are not perfect. Although blocking is avoided, the operations may become quite complex and can incur non-negligible computational overheads. More importantly, the algorithms all use multiple buffers to avoid corruption, so their space overhead is high, wasting memory resources that are severely limited in small, embedded systems.

In this paper, we present three new wait-free algorithms. We develop a generalized transformation mechanism that can improve existing wait-free algorithms by exploiting the temporal characteristics of communicating tasks, significantly reducing both space and execution time overheads. For some existing algorithms, we show up to 66% reduction in execution time and 70% reduction in memory requirements after applying our transformation. The transformed algorithms preserve all of the benefits of wait-free communication along with significant time and space savings.

In the following section, we present some background information and further motivate this work. We present our transformation mechanism in Section 3, and illustrate it using some actual IPC algorithms. Detailed evaluations are done in Section 4. We will put our work in the perspective of related work in

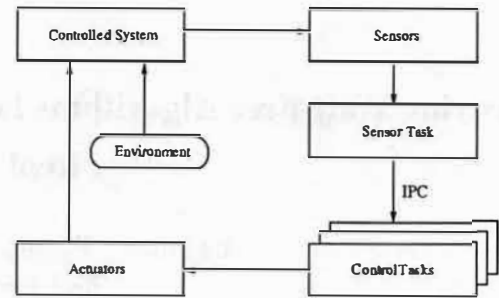


Figure 1: A schematic block diagram of a real-time system. Section 5, before concluding in Section 6.

## 2 Motivation

In this paper, we are primarily concerned with communication between a single writer and multiple readers. This is a very common scenario in embedded systems — ranging from as complex as automotive and industrial control systems to as simple as the controllers in kitchen appliances. Figure 1 shows a typical real-time system. The sensors are used to acquire information from the controlled system. A sensor task reads the data, performs any preprocessing, and distributes the information to the various control tasks. The control tasks perform computations and set the actuators based on this information, so it is important that they obtain uncorrupted, most-recently produced data from the sensor task.

Traditionally, the writer (i.e., sensor task) must pass the data to the readers (i.e., control tasks) by means of mailboxes, one of which is associated with each reader. However, if there is a large disparity in the execution frequencies of the tasks, especially if the sensor read rate is higher than the actuator control output rates, as is common, data messages will queue up in the mailboxes. The reader will obtain outdated messages, and will either have to process these or discard them to acquire the most current information. Generating multiple copies of each message incurs overheads in processor cycles and memory space, both of which are scarce resources in an embedded system. Therefore, the mailbox approach is neither appropriate nor efficient for typical IPC needed in real-time and embedded systems.

State messages are used to alleviate such problems. They were proposed in the MARS project [16] and implemented in ERCOS [25]. The state messages approach associates mailboxes with the writer instead of the readers, so only the writer associated with a particular mailbox can write to it. Furthermore, each message is assumed to include all data that needs to be communicated, so that the single, most current message conveys all information. Since data are time-sensitive, a new message can simply overwrite the previous one, effectively presenting the readers with the most up-to-date information. However, since the writer and readers can access the writer's mailbox concurrently, the readers can potentially read corrupted data if

<sup>1</sup> A concurrent object implementation is non-blocking if at least one process that is accessing the object can complete an operation within a finite number of steps regardless of failures. Furthermore, it is wait-free if every process that is accessing the object can complete an operation within a finite number of steps [13]. Wait-free is a stronger form of non-blocking as it ensures starvation-free access.

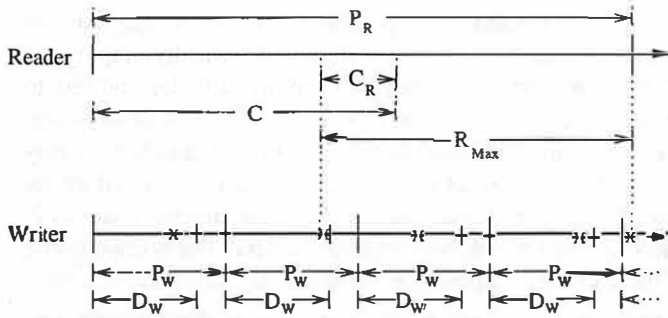


Figure 2: Reader and writer execution timelines, and each  $\times$  denotes a write operation performed by the writer.

the writer simultaneously writes new data.

There are many synchronization-based algorithms [9, 10] designed to ensure that reader tasks will always access uncorrupted messages. As mentioned earlier, synchronization, particularly with locks, can cause many problems of its own. Therefore, in this paper, we focus on wait-free, single-writer, multiple-reader IPC algorithms [7, 8, 17, 24, 31]. However, these algorithms have higher space overheads than the synchronization-based algorithms. Even though the worst-case time overhead of these algorithms is significantly lower than that of the synchronization-based ones, the execution overheads can still be significant. Later in this paper, we present a transformation mechanism that takes advantage of the real-time properties of the communicating tasks to reduce both the time and space overheads of this class of algorithms. First, however, we present a brief overview of real-time systems and tasks in the next section.

## 2.1 Attributes of Real-Time Tasks

Tasks in a typical real-time system are periodically invoked/released and executed.<sup>2</sup> Each task  $T$  is associated with various attributes, including its period  $P$ , relative deadline  $D$ ,<sup>3</sup> and worst-case execution time (WCET)  $C$ . The task must be run once each period, and needs to receive enough processing time to complete execution by its relative deadline. The real-time scheduler uses these attributes to decide when to run tasks, and can guarantee that all tasks will meet their deadlines as long as they require no more than their specified WCETs. From high-level program flow analysis and low-level timing information, a task's WCET can be determined statically. Figure 2 shows the relationship between these values for a typical scenario with one reader and one writer processes. The top timeline represents the reader's period. For simplicity, the reader's relative deadline is assumed to be equal to its period in our discussion and not shown here. In general, it is less than or equal to  $P_R$ , where  $P_R$  is the reader's period.  $C$  denotes the reader's WCET, and  $C_R$  is the time to perform a read operation.

<sup>2</sup> Aperiodic tasks can be handled by a periodic server [18], so the periodic task model is not a limiting assumption.

<sup>3</sup> This equals the deadline minus the release time of the task.

tion.  $R_{Max}$  represents the maximum time the reader can take to perform a read operation. Note in Figure 2 that the read operation is placed at the end of the reader task's execution. It is only drawn there to show the relationship between  $R_{Max}$ ,  $C_R$ ,  $P_R$  and  $C$  more clearly, but, in general, the read operation can be anywhere within the reader's execution time  $C$ . The bottom timeline represents 4 writer periods. The writer's period and relative deadline are denoted by  $P_W$  and  $D_W$ , respectively.

## 2.2 Temporal Concurrency Control

Since  $R_{Max}$  includes the time the reader is preempted by higher-priority tasks, it determines the maximum time the writer process may interfere with the reader within the reader's period without the reader missing its deadline.  $R_{Max}$  is calculated as follows:

$$R_{Max} = P_R - (C - C_R).$$

Assuming that all deadlines are met, Figure 2 illustrates the worst-case scenario in terms of the maximum number of preemptions of the reader by the writer task. This occurs when the first interfering-write happens as late as possible within the writer's period (first vertical dotted line — just before the writer's deadline) and the last interfering-write happens as early as possible within the writer's period (second vertical dotted line — just after the writer is released).

Let  $N_{Max}$  denote the maximum number of times the writer might interfere with the reader process during a read operation.  $N_{Max}$  can be calculated as:

$$N_{Max} = \max \left( 2, \left\lceil \frac{R_{Max} - (P_W - D_W)}{P_W} \right\rceil + 1 \right).$$

Therefore, if we use an  $(N_{Max} + 1)$ -deep circular buffer instead of a single message buffer, the writer can post messages cyclically without ever interfering with the reader process, assuming that the real-time constraints are met. This allows the reader and writer to access the message area independently of each other without blocking, using only temporal characteristics guaranteed by the real-time scheduling and a sufficiently-deep circular buffer to manage concurrency. With multiple readers, we simply choose an  $N_{Max}$  value large enough to work for all readers, i.e., compute it using the task with largest  $R_{Max}$ . Finally, we keep a pointer to the most recently written message. This is updated by the writer, and subsequently used by the readers to retrieve the latest message. This concept was first introduced in [16] and later implemented in the Non-Blocking Write (NBW) protocol [17].

This algorithm is very efficient in terms of execution time, i.e., almost as fast as using global variables with no protection. The only overhead associated with this algorithm is the cost of maintaining the pointer for the most recently written message. Therefore, it is easy to see that it has optimal timing behavior among wait-free algorithms.

## 2.3 Restricting Memory Use

With a deep enough buffer, the above algorithm will always guarantee that the readers will not acquire corrupted data. However, when  $R_{Max}$  is large or  $P_W$  is small,  $N_{Max}$  can get quite large and would require a large buffer space. This is undesirable, especially in embedded systems where memory is usually a scarce resource.

EMERALDS's state message algorithm [35] improves upon the NBW protocol. To limit memory usage, EMERALDS simply sets a static maximum buffer threshold for the state message. The reader tasks are divided into two groups, *fast* and *slow* readers. Tasks that have  $N_{Max}$  values less than this maximum buffer threshold are classified as fast readers, while the others are classified as slow readers.

The fast readers execute according to the NBW protocol. Since these readers have small  $N_{Max}$  values, they are both time- and space- efficient. For slow readers, EMERALDS provides a system call mechanism that (i) disables interrupts, (ii) copies the message from the shared buffer to the slow reader's local space on behalf of the reader, and (iii) re-enables interrupts. The overhead of this system call is quite high; however, according to the definition of slow readers, this call is invoked relatively infrequently, so it was claimed not to greatly impact the overall average-case execution time overheads.

As we will see in Section 4, the amount of overhead due to this system call is significant enough to make its average-case execution time much higher than the non-blocking algorithms. We would like to have the low execution overheads of the NBW protocol and the low memory usage achieved by the EMERALDS implementation, but without resorting to locks, disabled interrupts, or other synchronization-based concurrency control mechanisms. The following section details how to achieve this by transforming existing wait-free IPC mechanisms.

## 3 Improving Wait-Free IPC

In order to gain the benefits of wait-free IPC along with low memory usage, and low average- and worst- case execution times, we first generalize the concept of fast and slow readers (to reduce the memory requirements) introduced in EMERALDS. We then devise a transformation mechanism that can be applied to existing wait-free algorithms, preserves all of their inherent benefits, and simultaneously improves their performance.

Here, fast readers are defined as those tasks for which temporal concurrency control suffices to ensure uncorrupted reads without excessive memory usage. Slow readers consist of all of the other reader tasks, which would require too much memory to employ temporal concurrency control alone. The actual division of tasks would depend on the requirements of the final system, as we will see later.

We can transform IPC algorithms to use this concept of fast and slow readers. The fast readers will basically employ the NBW read mechanism, and will require sufficient buffers to ensure temporal concurrency control. The slow readers will use the existing IPC mechanism, although slight changes may be required because of the parallel approach employed by the fast readers. The writer requires more significant changes in order to interact with both types of readers. The precise nature of these changes depends on the actual algorithm transformed.

In general, we can make some predictions about the resulting performance. First, the average-case execution time (ACET) will decrease, since the highest-frequency readers will use the very efficient NBW mechanism. Worst-case execution time (WCET) is also often reduced, since for most algorithms, execution time depends on the number of simultaneous readers using the mechanism, which is reduced to only the slow readers. With the proper division of tasks into fast and slow readers (Section 3.4), the transformed algorithm should require much less memory on average than the original algorithm, and in the worst case, require no more than the original.

Our transformation mechanism can be illustrated more concretely by showing how we apply it to some actual algorithms. We first apply our transformation to the algorithm proposed by Chen *et al.* in [7]. We then show how to transform the Double Buffer algorithm, which we have developed and present in Section 3.2. Chen's algorithm has a relatively high execution time overhead and low space overhead, so we expect our transformation to primarily improve execution time. In contrast, the Double Buffer algorithm has a high space overhead and low execution time overhead. We expect this algorithm to benefit primarily from memory usage reduction after transformation. The following subsections detail the improved algorithms, which are evaluated in Section 4.

### 3.1 Improving Chen's Algorithm

Chen *et al.* [7] proposed a single-writer, multiple-reader wait-free algorithm using the Compare-And-Swap (CAS) instruction. This instruction is used to atomically modify the states of control variables used to ensure that the writer never writes to a buffer currently in use by some readers. The CAS instruction is commonly used in non-blocking algorithms to coordinate accesses to shared buffers and is supported on most modern microprocessors. Even if an architecture does not support this instruction, it can be synthesized by using other system primitives or system support [5]. The instruction  $CAS(A,B,C)$  is defined to be equivalent to atomically executing "if A equals B, then set A to C and return true, else return false."

Chen's algorithm requires  $(P + 2)$  message buffers, where  $P$  is the number of reader tasks. There is a global variable, *Latest*, that indexes to the most recently written message buffer. Additionally, each reader has an entry in a usage array indicating the buffer it is using. When the reader reads, it



```

int NSReader;           # Number of slow readers
int NBuffer;            # Number of buffers
int Latest;             # Index to the latest message
message Buff[NBuffer];  # Message buffer
char Reading[NSReader]; # Usage count

SlowReader() {
1:   Reading[i] = NBuffer;
2:   ridx = Latest;
3:   CAS( Reading[i], NBuffer, ridx );
4:   ridx = Reading[i];
5:   read Buff[ridx];
}

int GetBuff() {
  boolean InUse[NBuffer];
  for (i = 0; i < NBuffer; i++) InUse[i] = false;
  InUse[Latest] = true;
  for (i = 0; i < NSReader; i++) {
    j = Reading[i];
    if (j != NBuffer) InUse[j] = true;
  }
  for (i = ((Latest + 1) mod NBuffer); ;
12:   i = ((i + 1) mod NBuffer)) {
13:   if (InUse[i] == false)
14:     return i;
15:   }
}

Writer() {
16:   widx = GetBuff();
17:   write Buff[widx];
18:   Latest = widx;
19:   for (i = 0; i < NSReader; i++)
20:     CAS(Reading[i], NBuffer, widx);
}

```

Figure 3: Improved Chen's Algorithm.

first clears its entry, and then uses CAS to atomically set this to Latest if it is still cleared. It then reads back the value from its entry, and can then safely read from the indicated buffer. The writer has slightly more work to do. It first scans the usage array and selects a free buffer. It performs the write, updates Latest, and then must scan and set each reader entry that is cleared to Latest using CAS. This has been proven to ensure correct non-blocking IPC behavior in [7].

By taking into account the real-time properties of the communicating tasks, we can divide the reader set into two sets: fast and slow reader sets. By separating the reader set, we can reduce the space requirement from  $P + 2$  to  $M + \max(2, N)$ , where  $M$  is the number of slow readers and  $N$  is the number of buffers needed by the fast readers. Section 3.4 describes how to compute  $M$  and  $N$  in order to optimize for space. Because  $N$  is chosen to be less than, or equal to, the number of fast readers (i.e.,  $N \leq P - M$ ), the improved algorithm requires no more buffer space than the original algorithm. In the worst case (i.e., all readers are slow readers), the improved algorithm simply degenerates to the original algorithm. Furthermore, the execution time overheads will be greatly reduced, since fast readers use the very efficient NBW mechanism and the writer overhead is linear to the number of slow readers only, rather than all readers. Therefore, both space and time overheads can be reduced.

The Improved Chen's algorithm is shown in Figure 3. NSReader is the number of slow readers. NBuffer is the total number of message buffers. Buff[] is the array of mes-

sage buffers shared between the writer and readers. Latest is a control variable that indexes this array, indicating the most recently written message buffer. Reading[] is the usage array associated with the slow readers such that Reading[i] indicates which buffer entry the  $i^{th}$  slow reader is currently reading.

The slow readers operate identically to the readers in Chen's algorithm. Just before the  $i^{th}$  slow reader reads from the message buffer, Reading[i] is set to a value between 0 and NBuffer-1 to indicate the index of the buffer it will be reading. The writer will not overwrite this buffer slot as long as the slow reader is still using it. The slow reader first assigns Reading[i]=NBuffer to indicate that it is preparing to make a read operation. Then, it reads Latest, and attempts to set Reading[i] to this value atomically using CAS. If the writer has preempted the reader and completed a buffer write before this instruction, it would have already set Reading[i] to the new Latest value, and the reader's CAS would fail. In any case, by line 3, Reading[i] would have been atomically set to a buffer index that the writer will not use. So the slow reader simply reads the index and can now read from the indicated buffer safely.

The fast reader (not shown) is the same as in the NBW protocol. It relies only on temporal concurrency control, so it just reads Latest and uses the indicated buffer.

The Writer() process looks just like the one in Chen's algorithm. It calls GetBuff() to determine which buffer slot is safe to use next. After it writes the next message, it updates Latest and then modifies each Reading[i] using CAS if necessary.

The key difference lies in GetBuff() function, which is modified to allow temporal concurrency control for fast readers. First, to prevent the writer from interfering with slow readers, GetBuff() picks a buffer,  $m$ , such that no slow reader is using it (i.e., for all  $i$ , Reading[i]  $\neq m$ ). To protect the fast readers, as with the NBW protocol, we must ensure that there are at least  $(N - 1)$  writes between two consecutive writes to any particular buffer, where  $N$  is the buffer depth required for temporal concurrency control (Section 2.2). GetBuff() prevents the writer from interfering with the fast readers by cyclically choosing buffer entries starting from Latest. When NBuffer is chosen correctly (Section 3.4), even if each slow reader is using a unique buffer, there will be enough buffers (i.e., NBuffer - NSReader) left so that the cyclic selection will ensure sufficient time between two consecutive writes to the same buffer, satisfying the requirements for temporal concurrency control. Thus, the writer will not interfere with either fast or slow readers.

Let us illustrate this using the example shown in Figure 4. Assume that there are 20 readers, of which 3 are identified as slow readers. Assume further that relative execution frequencies of the fast readers and the writer are such that they require a 4-deep buffer to ensure temporal concurrency control. In this



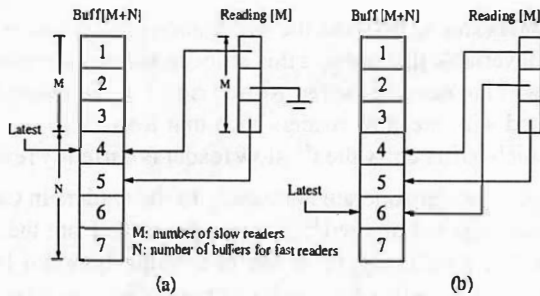


Figure 4: An example for Improved Chen's algorithm.

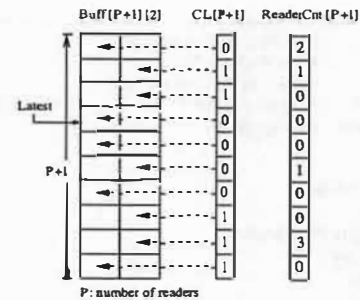


Figure 5: Constructs in the Double Buffer algorithm.

system, therefore, we need 7 message buffers (4 for the fast readers, and 1 for each of the slow readers), as compared to 22 buffers needed with the original Chen's algorithm. Figure 4(a) shows a particular execution state of the task set with `Latest` points to the 4<sup>th</sup> buffer slot. Since `Reading[0]` and `Reading[2]` point to the 4<sup>th</sup> and 5<sup>th</sup> buffer slots, the writer knows these may be in use, and will not use these buffers. Instead, it will cyclically select and write to the next available slot after `Latest`, the 6<sup>th</sup> buffer. The worst-case scenario occurs when the last slow reader now makes a read operation. It will now prevent the writer from using the 6<sup>th</sup> buffer. Even if the three slow readers never relinquish their buffers, the writer can continue to write cyclically to the remaining 4 buffers, with the repeating access pattern  $\{7, 1, 2, 3, 7, \dots\}$ . This ensures that no buffer is used more frequently than every fourth write, satisfying the conditions for the fast readers.

The biggest drawback of Chen's algorithm lies in the complexity of the `GetBuff()` function and the expensive CAS instruction itself. As shown in Figure 3, there are three loops inside of this function. The first one loops `NBuffer` times, and the second one loops `NSReader` times. Finally, the last one can potentially loop `NBuffer` times again. Furthermore, the writer has a loop that executes CAS `NSReader` times. As the number of slow readers decreases, we expect the performance enhancement from the Improved Chen's algorithm, as compared to the original Chen's algorithm.

### 3.2 Double Buffer Algorithm

We have devised a new wait-free IPC mechanism that is less computationally complex than Chen's algorithm. It, however, trades off time for space complexity, requiring approximately twice the buffer space. Hence, it is called the *Double Buffer* algorithm.

The basic constructs of the Double Buffer algorithm are shown in Figure 5, and the algorithm is summarized in Figure 6. A two-dimensional shared message buffer, `Buff[ ][ ]`, has  $(P+1)$  rows, where  $P$  is the number of reader tasks. Each row has two buffers. Associated with each row  $i$  is a usage count, `ReaderCnt[i]`, representing the number of readers currently using either buffer in the row, and a flag, `C1[i]`, indicating which of the two buffers is more current. A variable,

`Latest`, points to the row containing the most recently written data. A reader task first reads `Latest`, and indicates it is using the row by incrementing the usage count. It then reads the buffer indicated by the row's `C1` flag, and decrements the row's usage count when it finishes reading. Note that the increment and decrement operate directly on memory variables and must be atomic. This is commonly available on modern processors, including the x86 architecture.

The writer is fairly straightforward. It first scans `ReaderCnt[ ]`, and selects a row that is not being used by the readers. It then writes to the buffer that was least recently written in the selected row (i.e., opposite to the one indicated by the row's `C1` flag). We will see why this is necessary shortly. Finally, it updates the row's `C1` flag to point to the newly-written buffer, and sets `Latest` to the row that contains this buffer. In case each reader is concurrently reading from a unique row, this algorithm requires  $(P+1)$  rows for the writer to work correctly, where  $P$  is the number of readers. As each row has 2 buffers, the space required for the message buffer array is  $2(P+1)$ .

To see the correctness of the algorithm, let us consider the possible interference scenarios. The writer can only interfere with the reader when they both choose to use the same row. This can only occur in two cases. The first case can occur when a reader is interrupted after it has chosen a row (after line 1), but before it updates the use count (before line 2). The writer then executes, and can potentially choose the same row as the reader. The second case occurs when the writer is interrupted after it has chosen a row (after line 7). If this row happens to be `Latest`, then the reader can also choose to read from this same row. So, it is possible for the readers and the writer to select the same row  $i$ . However, the reader will read from the buffer indicated by `C1[i]`, while the writer will use the opposite one. As the writer updates `C1[i]` only after the complete message is written, and the reader always increments the use count before reading `C1[i]`, we can guarantee that the writer and readers cannot interfere with each other in this algorithm, even if they happen to use the same row.

The Double Buffer algorithm is less computationally complex than Chen's algorithms, but has a space requirement twice that of the original Chen's algorithm. In the next section, we use our transformation technique to improve the Double Buffer

```

int NReader;           # Number of readers
int NRows = NReader + 1; # Number of rows in the message buffer
int Latest;            # Index to the row with the latest message
message Buff[NRows][2]; # Message buffer
int ReaderCnt[NRows];  # Reader count for each row
boolean Cl[NRows];     # Column with more up-to-date message

```

```

Reader_i() {
1:  ridx = Latest;
2:  inc ReaderCnt[ridx];
3:  cl = Cl[ridx];
4:  read Buff[ridx][cl];
5:  dec ReaderCnt[ridx];
}

Writer() {
6:  for (i = Latest; ; i++)
7:    if (ReaderCnt[i mod NRows] == 0) break;
8:  cl = not Cl[i];
9:  write Buff[i][cl];
10: Cl[i] = cl;
11: Latest = i;
}

```

Figure 6: Double Buffer algorithm.

algorithm. As we will see in Section 4, the number of buffers required by the transformed Double Buffer algorithm is usually comparable to, if not less than, the original Chen’s algorithm.

### 3.3 Improved Double Buffer Algorithm

Applying the same techniques used in devising the Improved Chen’s algorithm, we now try to improve the Double Buffer algorithm. Again, we divide the reader tasks into fast and slow readers. The fast readers need a minimum of  $N$  buffers to ensure temporal concurrency control, while the  $M$  slow readers use the original Double Buffer scheme. The total message buffer requirements will now be  $2(M + \max(1, \lceil \frac{N}{2} \rceil))$  buffers, which is less than or equal to the original algorithm’s  $2(P + 1)$  buffers, assuming correct partitioning of the readers (see Section 3.4). As before, the highest-frequency readers now use the very low overhead NBW read mechanism, so execution times should be improved as well.

The data structures and algorithm for Improved Double Buffer are shown in Figures 7 and 8, respectively. The slow readers are unmodified from the original readers. Fast readers simply read from the buffer indicated by `Latest` and the corresponding row’s `Cl` entry. The writer, too, is mostly unmodified. To ensure temporal concurrency control for the fast readers, the writer should not reuse any particular buffer until at least  $N - 1$  subsequent writes have occurred. This is ensured by changing the buffer selection loop to search starting at row  $(\text{Latest} + 1) \bmod \text{NRows}$ . The rows are used cyclically, and the buffers within a row alternate on subsequent writes, so  $\lceil \frac{N}{2} \rceil$  rows suffice to ensure temporal concurrency control for the fast readers. Therefore, the improved algorithm needs  $2(M + \max(1, \lceil \frac{N}{2} \rceil))$  buffers.

To illustrate overhead improvements, let us consider a system with 20 reader tasks, of which 5 are classified as slow readers. Assume further that based on the  $N_{Max}$  calculations (Section 2.2), the fast readers need 7 buffers to ensure tempo-

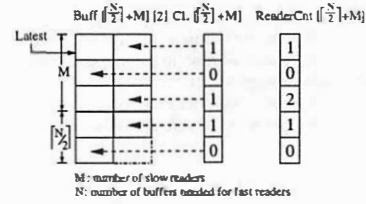


Figure 7: Constructs in the Improved Double Buffer algorithm.

ral isolation from the writer. With Improved Double Buffer, we need 18 message buffer slots, while the original needs 42, a significant memory reduction. Moreover, the other control variables are proportional to the number of rows, so they, too, are reduced. With the new algorithm, the slow readers and the writer remain virtually unchanged, but the fast readers have less computation than the original readers, so the overall execution overheads will decrease as well. Generally, as the number of fast readers increases, the execution performance increases, but this is not necessarily the case for space requirements. In the following section, we will determine how to partition a reader set into fast and slow readers, optimizing for space.

### 3.4 Identification of Fast Readers

We now present a simple algorithm for partitioning the reader set into fast and slow readers, optimizing for minimum memory usage. The algorithm is shown in Figure 9, and can be used with any single-writer, multiple-reader IPC scheme improved with our transformation by simply changing a few constants to match the algorithm.

The algorithm initially sets all reader tasks to be slow readers. It keeps the tasks sorted by non-decreasing order of their  $N_{Max}$  values, computed as with the NBW protocol (Section 2.2). It tries to move one task at a time from the slow reader set to the fast reader set, and recomputes the number of buffers needed,  $(S + F)$ , where  $S$  is the requirement for the slow readers, and  $F$  for the fast readers. By keeping track of the setting with lowest memory use so far, after a single pass through all of the tasks, we obtain the `Splitpoint`, which indicates the last fast reader. All tasks with lower  $N_{Max}$  values are also part of the fast reader set.

This partitioning of the reader set is optimal with respect to the number of message buffers. This is easy to show: take a partitioning that is space-optimal, and let task  $i$  be the fast reader with the largest  $N_{Max}$  value. Now, all tasks with lower  $N_{Max}$  values than task  $i$  must also be part of the fast reader set (otherwise, we can move them to the fast reader set; they will not affect the number of buffers needed for the fast readers (i.e., largest  $N_{Max}$  value), but will reduce the slow reader set’s buffer requirements, and the optimality assumption would be invalid). Since the above algorithm considers all partitions in which all tasks with less than a particular  $N_{Max}$  value are in the fast reader set, the optimal partition will be found by the algorithm.

```

int Latest;           # Index to the row with the latest message
int NRows;           # Number of rows in the message buffer
message Buff[NRows][2]; # Message buffer
int ReaderCnt[NRows]; # Reader count for each row
boolean Cl[NRows];    # Column with more up-to-date message

SlowReader_i() {
1:   ridx = Latest;
2:   inc ReaderCnt[ridx];
3:   cl = Cl[ridx];
4:   read Buff[ridx][cl];
5:   dec ReaderCnt[ridx];
}
FastReader_i() {
6:   ridx = Latest;
7:   boolean cl = Cl[ridx];
8:   read Buff[ridx][cl];
}
Writer() {
9:   i = (Latest + 1) mod NRows;
10:  for (; i = ((i + 1) mod NRows))
11:    if (ReaderCnt[i] == 0) break;
12:  cl = not Cl[i];
13:  write Buff[i][cl];
14:  Cl[i] = cl;
15:  Latest = i;
}

```

Figure 8: Improved Double Buffer algorithm.

The partitioning algorithm uses certain constants that depend on the specific IPC mechanism used. For the initialization, *Splitpoint* is always set to *NULL* and *F* always set to 0, but *MinNumBuff* and *S* are both set to the number of buffers needed assuming that all tasks are slow readers. For the Improved Chen’s algorithm, this is  $(P + 2)$ , and for the Improved Double Buffer, it is  $2(P + 1)$ , where *P* is the number of tasks. Additionally, *V* is the number of buffers used for each additional slow reader, and is set to 1 and 2, for Chen’s and the Double Buffer mechanisms, respectively.

We illustrate the partitioning algorithm using the sample task set in Figure 10, which indicates the writer’s period and relative deadline, as well as the readers’ periods (relative deadlines) and computation times.  $R_{Max}$  and  $N_{Max}$  values, assuming  $C_R$  is negligible and the readers’ relative deadlines are equal to their periods, are also shown. Assuming Double Buffer algorithm, initially  $S = \text{MinNumBuff} = 16$ ,  $F = 0$ , and all readers are in the slow reader set. Tasks are moved one at a time according to their  $N_{Max}$  values, so first, Reader 0 is moved to the fast reader set. Now  $S = 14$  and  $F = 3$ , so  $(F + S)$  is not the lowest value seen, and *Splitpoint* is not changed. We continue with Reader 1, resulting in  $S = 12$  and  $F = 3$ , so  $S + F \leq \text{MinNumBuff}$  holds. *Splitpoint* is updated to Reader 1, and *MinNumBuff* is set to  $S + F$ . We repeat this with all of the readers, in order. By the end, *Splitpoint* points to Reader 4, and *MinNumBuff* = 10. So, with the first five readers as fast readers, we achieve the minimum number of buffers required for this example, a 37.5% reduction from the original algorithm.

```

Order reader tasks by  $N_{Max}$  from smallest to largest;
# Note:  $S_0$  is the no. of buffers needed if all tasks are slow readers
 $S = S_0$ ;           # no. of buffers for slow readers
 $F = 0$ ;           # no. of buffers for fast readers
MinNumBuff =  $S_0$ ;
Splitpoint = NULL;

```

```

Foreach readertask  $T_R$  (ordered by  $N_{Max}$ )
  Move  $T_R$  from the slow reader set to the fast reader set;
   $S = V \times \text{sizeof}(\text{slow reader set})$ ;
   $F = T_R.s(N_{Max} + 1)$ ;
  if ( $S + F \leq \text{MinNumBuff}$ )
    Splitpoint =  $T_R$ ;
    MinNumBuff =  $S + F$ ;

```

Figure 9: Algorithm to find space-optimal division of fast and slow readers and the amount space required.

Process	$P_W$	$D_W$		
Writer	10	7		
	$P_R$	$C$	$R_{Max}$	$N_{Max}$
Reader 0	8	4	4	2
Reader 1	12	7	5	2
Reader 2	23	14	9	2
Reader 3	22	9	14	3
Reader 4	50	30	20	3
Reader 5	150	25	125	14
Reader 6	500	25	475	49

Figure 10: Task set with one writer and seven reader processes.

### 3.5 Transformation Mechanism

We have shown here how two different single-writer, multiple-reader wait-free IPC mechanisms can be modified to take into account real-time characteristics of tasks to reduce both memory and execution time overheads. In general, we can apply our transformation to other such IPC algorithms with the following steps.

- Step 1.** Identify fast and slow readers for a particular system: simply apply the algorithm in Section 3.4. This will minimize the number of message buffers needed, while still ensuring temporal isolation between the writer and the fast readers.
- Step 2.** Fine-tune reader sets: we may not always want to optimize for space, so we can adjust the partitioning obtained in Step 1 if needed.
- Step 3.** Convert reader code to slow reader code: Typically, there are no modifications needed for slow readers, so this is just a renaming step.
- Step 4.** Introduce fast reader code: The fast readers are trivially implemented — they just read the pointer indicating the most recently written message buffer, and then read from that buffer.
- Step 5.** Modify writer code to ensure temporal isolation with fast readers: this is the most significant change required.

Since most algorithms have some code for selecting a buffer to write, this step usually only requires modifying the selector to ensure that the same buffer is not reused within  $N$  consecutive writes. Sometimes, this can simply be done by using the available buffers in a cyclic fashion, and having enough total buffers.

Applying these steps, we can modify existing wait-free single-writer, multiple-reader algorithms to use real-time characteristics of the tasks and reduce processing and memory costs.

## 4 Performance Evaluation

The goal of our transformation mechanism is to reduce the time and space overheads when applied to single-writer, multiple-reader algorithms. We now evaluate how much improvement we can achieve with the proposed transformation. Specifically, we will compare a total of 9 different IPC mechanisms, including Chen's, Improved Chen's, Double Buffer, and Improved Double Buffer algorithms.

We also consider another wait-free, single-writer, multiple-reader IPC mechanism, Peterson's algorithm, as well as our transformed version of it. In Peterson's algorithm [24], the reader determines if its read is corrupted, and may have to perform the read up to 3 times. The writer may also have to write a message up to  $(P + 2)$  times, where  $P$  is the number of readers. The mechanism has been revised [34] such that readers read a message at most 2 times, and the writer writes a message at most  $(P + 1)$  times to avoid corruption. We only consider the revised version here. We derive the Improved Peterson's algorithm by applying our general transformation as described in Section 3.5.

For the purpose of comparison, we also evaluate the NBW protocol and the EMERALDS variant of this. As discussed earlier, NBW is the most efficient algorithm in terms execution time, but may induce high space overheads. The EMERALDS IPC mechanism tries to limit memory use at some cost to performance. Finally, we also include a very efficient implementation of synchronization-based IPC, using a lock algorithm that relies on the atomic Test-And-Set instruction, to show the trade-offs between synchronization-based and synchronization-free mechanisms.

To make fair and comprehensive comparisons between these algorithms, we have considered various parameters trying to answer the following questions.

- How much does the transformation reduce the average-case and worst-case execution times?
- How much does the transformation reduce the buffer space requirement?
- Is the transformation applicable in both uniprocessor and multiprocessor environments? How do they differ?

Class	Subclass	Percentage within Class	Relative Frequency to the Writer Process
Fast	Fastest	15–25%	twice as frequent
	Fast	75–85%	1–15 times less frequent
Slow	Slow	75–85%	15–50 times less frequent
	Slowest	15–25%	50–100 times less frequent

Figure 11: Reader task set distribution.

- Will different message sizes affect the results?
- Will the size of the reader set affect the results?

We evaluate the algorithms for memory usage and execution time overheads, in both average and worst cases, and for both uniprocessor and symmetric multiprocessor (SMP) environments. The only exception is for the EMERALDS IPC mechanism, which is evaluated only for uniprocessors. Because it assumes that operations are atomic if interrupts are disabled, it will not work correctly with SMP architectures where this assumption does not hold.

### 4.1 Experiment Setup

The algorithms we evaluate in this section are implemented and executed under EMERALDS OS [35] running on a Pentium-III 500Mhz processor. The experiments use a synthetic reader task set, which is divided into two sets — fast readers and slow readers, where 'fast' and 'slow' are defined relative to the writer's period. In a real system, there are usually tasks that are executed very frequently, and tasks that run very infrequently. To model this behavior, we further divide the fast and slow reader sets into finer-grained categories, as shown in Figure 11. By making approximately 20% of fast and slow readers either very fast or very slow, the resulting task set represents realistic range of task periods that may occur in a real-time embedded system. A random reader task set is generated for each experiment according to the desired division of readers into the four categories.

### 4.2 Average vs. Worst-case Execution Time

The average-case (ACET) and worst-case execution times (WCET) to perform an IPC read/write operation are both important factors in the performance of an IPC algorithm. A low ACET would indicate that the algorithm generally incurs low computation overheads. However, to provide timeliness guarantees in embedded real-time systems, the scheduler must account for the WCET. An algorithm with low ACET but high WCET may result in poor system utilization.

The ACET and WCET of the SMP versions of the eight evaluated algorithms are shown on the top and bottom rows, respectively, in Figure 12. The SMP versions of the algorithms include bus-lock operations to ensure the atomic operation of the critical CAS and TAS instructions with multiple processors. The message size is 8 bytes, and the task set consists

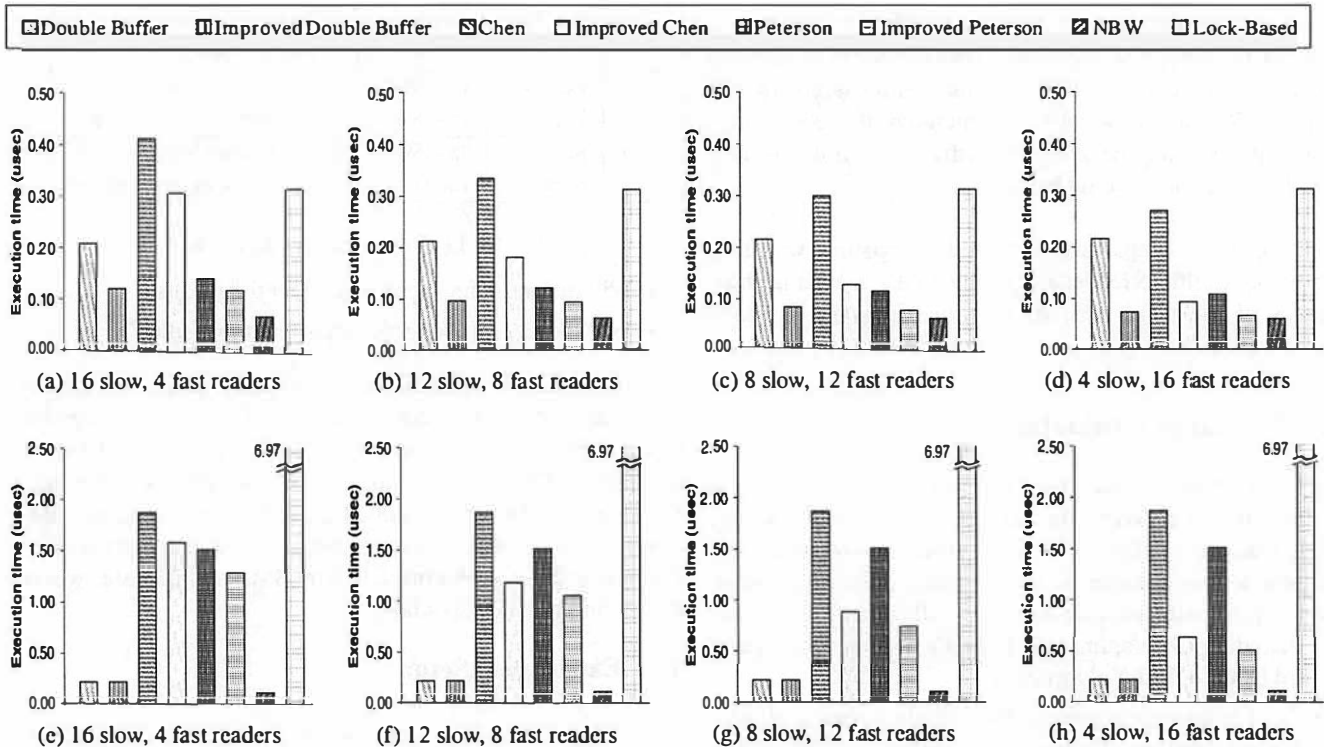


Figure 12: The top and bottom rows show the average-case and worst-case execution times, respectively, of the SMP version of the algorithms, to perform an IPC read / write operation with 8-byte message size.

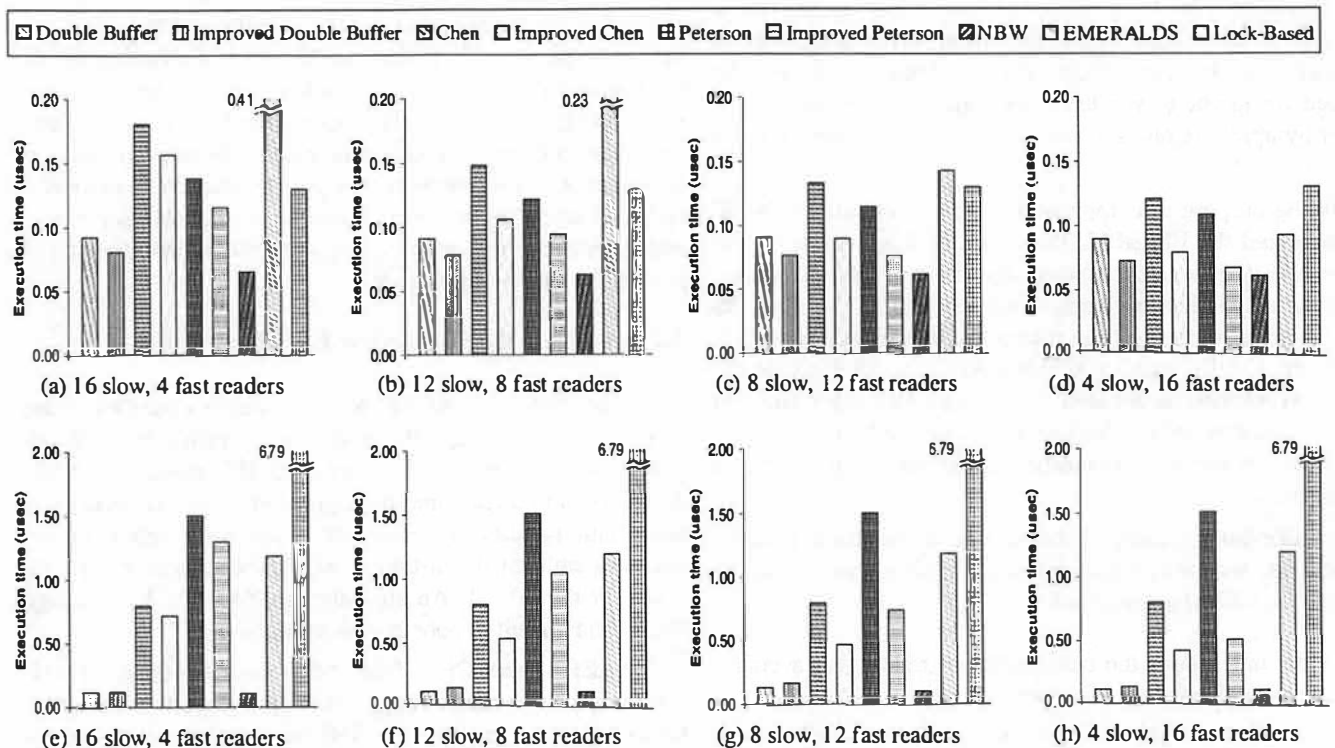


Figure 13: The top and bottom rows show the average-case and worst-case execution times, respectively, of the uniprocessor version of the algorithms, to perform an IPC read / write operation with 8-byte message size.



Double Buffer Improved Double Buffer Chen Improved Chen Peterson Improved Peterson NBW EMERALDS Lock-Based

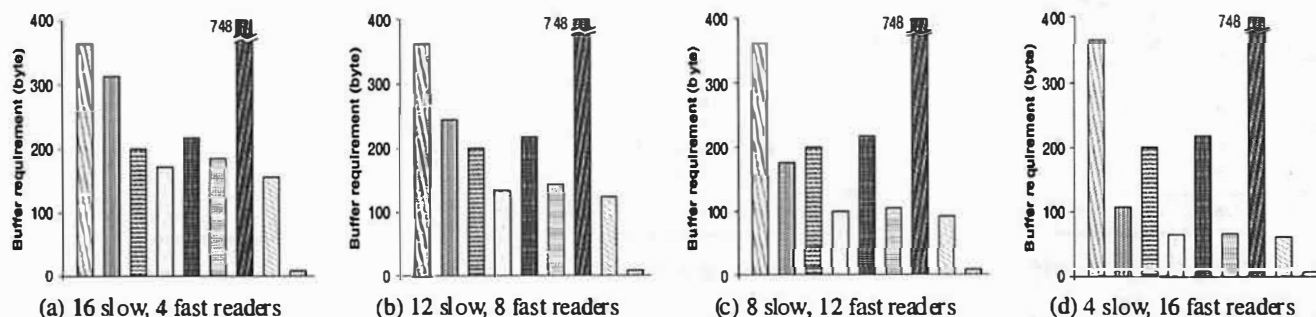


Figure 14: These graphs show the space requirements for different algorithms with 8-byte messages (note that the space requirement is architecture-independent).

of 1 writer and 20 readers, of which a varying fraction are fast readers. Specifically, we evaluated these algorithms when the reader set contains 20%, 40%, 60% and 80% fast readers. The first three pairs of columns on the graphs are for the three single-writer, multiple-reader algorithms and their corresponding transformed algorithms. We can see significant reductions in both the ACET and WCET from comparison of the transformed algorithms with the original ones. As the number of fast readers in the reader set increases, the reduction in computation time for the transformed algorithms gets more pronounced. ACETs for Double Buffer and Chen's algorithms improve by as much as 66%, and for Peterson's algorithm by as much as 38%. This trend is shown in Figure 15(a).

Although the amount of improvement is a non-decreasing function of the percentage of fast readers in the reader set, the magnitude of this improvement depends on the particular algorithm. In these experiments, all of the transformed algorithms perform better than the original versions except for the WCET of the Double Buffer algorithm. This can be attributed to the fact that the WCET for the Double Buffer algorithm occurs in the slow readers. As this time is not affected by the number of slow readers in the system, the WCET does not improve. For the other algorithms, the WCETs occur in the writers, whose overheads are functions of the number of slow readers, and, therefore, improve greatly.

It is interesting to note that even though the ACET of the lock-based algorithm is only up to 4 times larger than those of the transformed wait-free algorithms, its WCET is much higher — 4 to 30 times higher. This is in fact an underestimate of the true overhead of the lock-based mechanism, since we assume no blocking time here. In actual systems, unless the system employs mechanisms to limit blocking times, the lock-based execution time may be unbounded.

### 4.3 Uniprocessor vs. SMP

The correctness of some asynchronous algorithms rely on the fact that certain instructions will be executed atomically. For

example, Chen's algorithm requires that the CAS instruction be performed atomically. For SMP architectures, this requires that expensive bus-locking (e.g., by using the *LOCK* prefix in the x86 architecture) be performed to ensure an atomic read-modify-write of memory. Under uniprocessor environments, however, such measures are generally not needed. In most architectures, including x86, these instructions are already guaranteed to be atomic with respect to uniprocessor systems without incurring any additional overheads. As a result, we can reduce the costs of CAS for Chen's algorithm, atomic *inc* and *dec* for Double Buffer, and TAS for the lock-based mechanism. We now repeat the above experiments, but using code restricted to uniprocessor machines. The results, including evaluations of the EMERALDS IPC mechanism, are shown in Figure 13.

As expected, the ACET and WCET of these algorithms are lower than their counterparts for SMP. Even in this case, we can still save a significant percentage of execution time overheads. It is worth noting how close the ACETs of the transformed algorithms are to the optimal NBW protocol execution time. WCET improvements after transformation are even more pronounced than for ACET, except with the Double Buffer algorithm. This anomaly is due to the complexity we have introduced in the writer to handle both kinds of readers. Nonetheless, the WCET of the Double Buffer algorithm is still very close to that of NBW.

We summarize the reduction in ACET as the percentage of fast readers changes in Figure 15(b). Compared to the SMP results in Figure 15(a), the ACET reduction in uniprocessor environments is less pronounced. Nonetheless, our transformation still reduces a good amount in execution time.

### 4.4 Savings in Space

Thus far, we have shown that our transformation mechanism enhances the performance of algorithms in terms of the ACET and the WCET in both SMP and uniprocessor environments. Here, we present results to support our claim that the transformation mechanism not only reduces the time overheads but



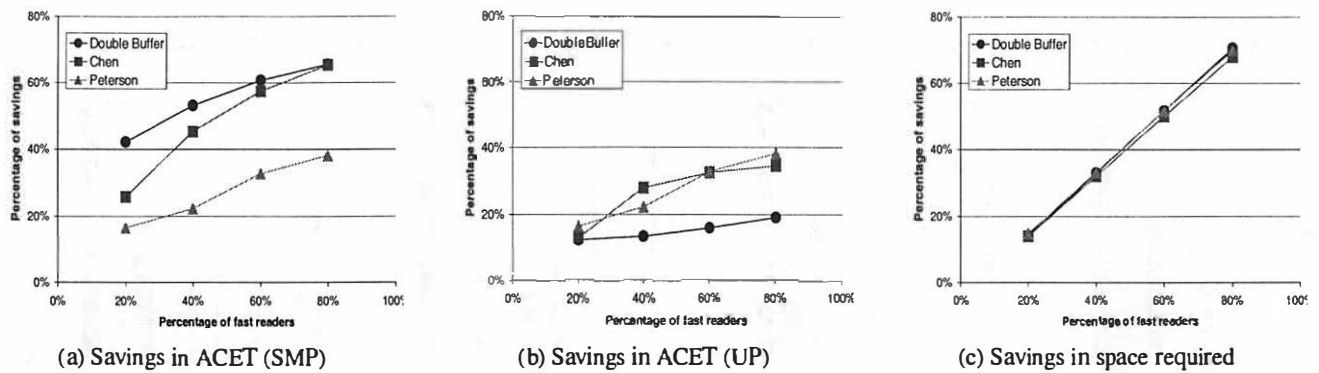


Figure 15: Varying the percentage of fast readers in the reader set, (a) and (b) show the percentage of savings in ACET for SMP and uniprocessor versions of the algorithms, respectively. (c) shows the percentage of savings in space.

also the space overheads of the algorithms. This is shown in Figure 14. Again, we varied the percentage of fast readers in the reader set. As expected, the amount of improvement increases as the number of fast readers increases. Moreover, the synchronization-based algorithm requires the least space, since it only needs a single shared message buffer. The NBW protocol and lock-based IPC, therefore, represent the extreme cases for the tradeoff between space requirements and WCET. The non-blocking IPC mechanisms, especially with our transformation, provide a good compromise, balancing WCET and memory usage.

Interestingly, the percentage of space reduction for all three transformed algorithms is the same, as shown in Figure 15(c). This does make sense since the memory requirements of the three original algorithms are all proportional to the number of readers in the reader set. So, the memory used by the transformed algorithms decreases proportionally to the number of slow readers. The slight variations in Figure 15(c) are due to some of the control variables that do not scale with the number of reader tasks. Overall, we achieve a reduction in memory usage that ranges from 14 to 70%.

#### 4.5 Effects of Message and Reader Set Size

The experiments in the previous sections all use 8-byte messages. To see how varying the message size affects the savings in time and space, we have performed the same set of experiments with larger messages (64 bytes). The measurements follow a similar trend, but the percentage reduction in execution time is less than when using 8-byte messages. This is because the execution overhead of the actual message buffer read/write operation, which cannot be reduced, becomes a more dominant part of the total execution overheads. The percentage reductions in space overheads are the same, or slightly better than for the 8-byte message case, since the constant overheads of some of the control variables are less apparent. Due to the substantially similar results, the 64-byte message measurements are not presented here.

We have also conducted experiments while varying the total

size of the reader set. Running the previous experiments with 10 reader tasks resulted in nearly identical relative performance improvements with our transformation mechanism. Of course, with fewer readers, any complexity increase in the writer task has greater weight in the average execution time, but this is offset by the performance gains in the fast readers. Space reduction, as before, is basically linear to the percentage reduction in the number of slow readers. Again, due to their substantially similar results, the data for the 10 readers case are omitted here.

## 5 Related Work

Some earlier work [17, 20] on lock-free objects was done using read-and-check loops. The reader is required to check if its reading was interfered with by the writer, in which case it performs the read operation again until it succeeds. Optimization techniques to reduce the number of loops were proposed in [15], using an exponential backoff policy. Kopetz *et al.* [17] and Anderson *et al.* [2] later demonstrated how to bound the number of retries by either increasing the buffer size or through judicious scheduling.

To reduce the time overheads associated with read-and-check loops, algorithms that make space and time tradeoffs were later proposed [6–8, 17, 24, 31, 35]. These algorithms provide a good middle-ground between the purely lock-based approach (high WCET) and the purely buffer-based approach (large buffer requirement). The benefit of these algorithms is that less time is wasted in read-and-check loops and the timing behavior is more predictable, improving schedulability of task sets as well as system utilization. Although the timing behavior is more predictable, the computational complexity of these algorithms is still high. Moreover, they may still incur a large buffer space requirement, and may be difficult to use in small-memory embedded systems. This difficulty can be overcome by our transformation mechanism, which makes significant reductions in both time and space overheads.

Most non-blocking algorithms rely on the availability of some form of atomic memory update instructions, such

as Compare-And-Swap or Load-Linked/Store-Conditional in hardware. A few modern hardware platforms, however, do not implement some of these instructions. The author of [23] demonstrated how to emulate these instructions by synthesizing more commonly-implemented instructions to close the gap between the primitives that the algorithm designers rely upon, and the primitives provided by the hardware. Bershad [5] proposed how to implement CAS instruction in software by using operating system support, and Greenwald *et al.* [12] generalized this technique to implement Double-Word CAS and Multi-Word CAS instructions. Similar work was done in Synthesis [22] and Cache kernel [12]. Our transformation mechanism does not use such operations, so it is not directly affected by whether the atomic operations used by the original IPC algorithms are supported by the hardware or are emulated. However, the degree of performance improvement will be different. All of the algorithms we evaluated use atomic update instructions supported natively by the x86 architecture. We expect an even greater improvement with our transformation if these instructions are emulated since the overheads for emulation will most likely be higher.

Herlihy [13] proposed the first general methodology to transform sequential data objects to the equivalent non-blocking structures. Alemany *et al.* [1] and LaMarca [19] proposed techniques to reduce the inefficiencies in applying this methodology to large objects at the cost of more communication between the application process and the operating system. Other methods to improve this were proposed in [4, 32]. Prakash *et al.* [26] and Turek *et al.* [32] presented techniques to transform multiple-lock concurrent objects into lock-free objects. However, it was shown that their transformed algorithms are less efficient than the corresponding lock-based algorithms [15, 19, 30]. These authors are concerned with transforming sequential objects to non-blocking objects, and the related performance issues. We take the next logical step by transforming non-blocking objects, in particular, those with single-writer, multiple-reader semantics, to better-performing and less space-consuming non-blocking objects.

Some interesting work [14, 15, 27] has also been done in the construction of more complex concurrent objects. Concurrent non-blocking array-based stacks, FIFO queues and multiple lists were implemented using Double-Compare-And-Swap in [12]. Valois introduced non-blocking algorithms for queues, linked-lists, and arrays in [33]. Eliot *et al.* [11] proposed non-blocking algorithms for garbage collection. We do not look at these complex structures, but focus instead on the more common, single-writer, multiple-reader state message construct, used for IPC in embedded systems.

## 6 Conclusions

In this paper, we have argued for efficient IPC mechanisms, particularly for memory- and processing-power- constrained

embedded real-time systems. Traditional and synchronization-based IPC methods incur too much time overhead and follow incorrect semantics for most of such systems. Instead, we considered wait-free, single-writer, multiple-reader IPC algorithms, which are more appropriate for these systems, but still can incur substantial overheads.

By taking advantage of the temporal characteristics of the tasks in these systems, we have proposed a general transformation mechanism that can significantly reduce both space and time overheads of the wait-free IPC algorithms. This allows the most frequently-executing reader tasks to use very low-overhead operations, while reducing the total number of buffers needed to ensure corruption-free message passing. We have demonstrated our transformation on the existing Chen's algorithm and the new Double Buffer algorithm that we have introduced here.

Our extensive experiments show a 17–66% reduction in ACET, and a 14–70% reduction in memory requirements for the IPC algorithms improved with our transformation. For algorithms with relatively high WCETs, these are shown to be improved greatly as well. The experiments also demonstrate the tradeoff between time and space in IPC mechanisms: the NBW protocol is time-optimal, but requires large buffers, while a lock-based approach requires just a single message buffer, but suffers from very high worst-case execution overheads. Overall, the single-writer, multiple-reader non-blocking algorithms are good intermediate solutions, balancing WCET and space requirements. With our transformation, we can do even better, reducing both time and space requirements of these algorithms.

This transformation mechanism can be applied to other non-blocking IPC algorithms that are not considered here, and make them better optimized for systems with real-time characteristics. In the future, we would like to extend our methodology to reduce synchronization overheads in more general IPC algorithms with multiple-writer semantics and to extend this to more general communication channels as well.

## 7 Acknowledgments

We would like to thank our shepherd, Carla Ellis, and the anonymous reviewers for their excellent feedback. We also like to thank John Reumann for some helpful discussions earlier in this work.

## References

- [1] J. Alemany and W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 125–134, 1992.
- [2] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 28–37, Dec 1995.

- [3] T. P. Baker. Stack-based scheduling of realtime processes. *RT-SYSTS: Real-Time Systems*, 3, 1991.
- [4] Greg Barnes. A method for implementing lock-free data structures. In *5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 93)*, 1993.
- [5] B. Bershad. Practical considerations for non-blocking concurrent objects. In *IEEE International Conference on Distributed Computing Systems*, pages 264–273, 1993.
- [6] James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.
- [7] J. Chen and A. Burns. A fully asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical Report YCS-288, Department of Computer Science, University of York, 1997.
- [8] J. Chen and A. Burns. A three-slot asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical Report YCS-186, Department of Computer Science, University of York, 1997.
- [9] P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. *Communications of the Association of Computing Machinery*, 14(10):667–668, 1971.
- [10] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the Association of Computing Machinery*, 8(9):569, 1965.
- [11] J. Eliot and B. Moss. Lock-free garbage collection for multiprocessors. In *Parallel Algorithms and Architectures*, 1991.
- [12] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Operating Systems Design and Implementation*, pages 123–136, 1996.
- [13] M. Herlihy. A methodology for implementing highly concurrent data structure. *Proceeding of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, 1989.
- [14] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [15] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.
- [16] H. Kopetz and *et al.* Distributed fault-tolerant real-time systems: the Mars approach. *IEEE Micro*, 9(1):25–40, 1989.
- [17] H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronisation problem. In *IEEE Real-Time Systems Symposium*, pages 131–137, 1993.
- [18] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.
- [19] Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. In *ACM Symposium on Principles of Distributed Computing*, pages 130–140, 1994.
- [20] L. Lamport. Concurrent reading and writing. *Communications of ACM*, 20(11):806–811, Nov 1977.
- [21] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
- [22] H. Massalin and C. Pu. A lock-free multiprocessor os kernel. Technical Report CUCS-005-91, Department of Computer Science, Columbia University, 1991.
- [23] M. Moir. Practical implementations of non-blocking synchronization primitives. In *ACM Symposium on Principles of Distributed Computing*, pages 219–228, 1997.
- [24] G. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.
- [25] S. Poledna, T. Mocken, and J. Schiemann. Ercos: an operating system for automotive applications. Technical Report 960623, Society of Automotive Engineers Technical Paper Series, 1996.
- [26] S. Prakash, Y.-H. Lee, and T. Johnson. Non-blocking algorithms for concurrent data structures. Technical Report 91-002, Department of Computer Science, University of California, Los Angeles, 1991.
- [27] S. Prakash, Y.-H. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, 1994.
- [28] R. Rajkumar. Synchronization in real-time systems - a priority inheritance approach. *Kluwer Academic Publishers*, 1991.
- [29] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.
- [30] Nir Shavit and Dan Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.
- [31] H. R. Simpson. Four-slot fully asynchronous communication mechanism. In *IEEE Proceedings*, 1990.
- [32] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *ACM Proceedings of the Principles of Database Systems*, pages 212–222, 1992.
- [33] J. D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Las Vegas, NV, 1994.
- [34] K. Vidyasankar. Concurrent reading while writing revisited. *Distributed Computing*, 4(2):81–86, 1990.
- [35] K. Zuberi, P. Pillai, and K. G. Shin. EMERALDS: a small-memory real-time microkernel. In *ACM Symposium on Operating Systems Principles*, volume 34, pages 277–299, 1999.

# Robust Positioning Algorithms for Distributed Ad-Hoc Wireless Sensor Networks

Chris Savarese   Jan Rabaey  
*Berkeley Wireless Research Center*  
{savarese,jan}@eecs.berkeley.edu

Koen Langendoen  
*Faculty of Information Technology and Systems*  
*Delft University of Technology, The Netherlands*  
koen@pds.twi.tudelft.nl

## Abstract

A distributed algorithm for determining the positions of nodes in an ad-hoc, wireless sensor network is explained in detail. Details regarding the implementation of such an algorithm are also discussed. Experimentation is performed on networks containing 400 nodes randomly placed within a square area, and resulting error magnitudes are represented as percentages of each node's radio range. In scenarios with 5% errors in distance measurements, 5% anchor node population (nodes with known locations), and average connectivity levels between neighbors of 7 nodes, the algorithm is shown to have errors less than 33% on average. It is also shown that, given an average connectivity of at least 12 nodes and 10% anchors, the algorithm performs well with up to 40% errors in distance measurements.

## 1 Introduction

Ad-hoc wireless sensor networks are being developed for use in monitoring a host of environmental characteristics across the area of deployment, such as light, temperature, sound, and many others. Most of these data have the common characteristic that they are useful only when considered in the context of where the data were measured, and so most sensor data will be stamped with position information. As these are ad-hoc networks, however, acquiring this position data can be quite challenging.

Ad-hoc systems strive to incorporate as few assumptions as possible about characteristics such as the composition

of the network, the relative positioning of nodes, and the environment in which the network operates. This calls for robust algorithms that are capable of handling the wide set of possible scenarios left open by so many degrees of freedom. Specifically, we only assume that all the nodes being considered in an instance of the positioning problem are within the same connected network, and that there will exist within this network a minimum of four anchor nodes. Here, a connected network is a network in which there is a path between every pair of nodes, and an anchor node is a node that is given *a priori* knowledge of its position with respect to some global coordinate system.

A consequence of the ad-hoc nature of these networks is the lack of infrastructure inherent to them. With very few exceptions, all nodes are considered equal; this makes it difficult to rely on centralized computation to solve network wide problems, such as positioning. Thus, we consider distributed algorithms that achieve robustness through iterative propagation of information through a network.

The positioning algorithm being considered relies on measurements, with limited accuracy, of the distances between pairs of neighboring nodes; we call these range measurements. Several techniques can be used to generate these range measurements, including time of arrival, angle of arrival, phase measurements, and received signal strength. This algorithm is indifferent to which method is used, except that different methods offer different tradeoffs between accuracy, complexity, cost, and power requirements. Some of these methods generate range measurements with errors as large as  $\pm 50\%$  of the measurement. Note that these errors can come from multiple sources, including multipath

interference, line-of-sight obstruction, and channel inhomogeneity with regard to direction. This work, however, is not concerned with the problem of determining accurate range measurements. Instead, we assume large errors in range measurements that should represent an agglomeration of multiple sources of error. Being able to cope with range measurements errors is the first of two major challenges in positioning within an ad-hoc space, and will be termed the *range error problem* throughout this paper.

The second major challenge behind ad-hoc positioning algorithms, henceforth referred to as the *sparse anchor node problem*, comes from the need for at least four reference points with known location in a three-dimensional space in order to uniquely determine the location of an unknown object. Too few reference points results in ambiguities that lead to underdetermined systems of equations. Recalling the assumptions made above, only the anchor nodes will have positioning information at the start of these algorithms, and we assume that these anchor nodes will be located randomly throughout an arbitrarily large network. Given limited radio ranges, it is therefore highly unlikely that any randomly selected node in the network will be in direct communication with a sufficient number of reference points to derive its own position estimate.

In response to these two primary obstacles, we present an algorithm split into two phases: the *start-up* phase and the *refinement* phase. The start-up phase addresses the sparse anchor node problem by cooperatively spreading awareness of the anchor nodes' positions throughout the network, allowing all nodes to arrive at initial position estimates. These initial estimates are not expected to be very accurate, but are useful as rough approximations. The refinement phase of the algorithm then uses the results of the start-up algorithm to improve upon these initial position estimates. It is here that the range error problem is addressed.

This paper presents our algorithms in detail, and discusses several network design guidelines that should be taken into consideration when deploying a system with such an algorithm. Section 2 will discuss related work in this field. Section 3 will elaborate our two-phase algorithm approach, exploring in depth the start-up and refinement phases of our solution. Section 4 will discuss some subtleties of the algorithm in relation to our simulation environment. Section 5 reports on the experiments performed to characterize the performance of our algorithm. Finally, Section 6 is a discussion of design guidelines and algorithm limitations, and Section 7 concludes the paper.

## 2 Related work

The recent survey and taxonomy by Hightower and Borriello provides a general overview of the state of the art in location systems [7]. However, few systems for locating sensor nodes in an ad-hoc network are described, because of the aforementioned range error and sparse anchor node problems. Many systems are based on the attractive option of using the RF radio for measuring the range between nodes, for example, by observing the signal strength. Experience has shown, however, that this approach yields very inaccurate distances [8]. Much better results are obtained by time-of-flight measurements, particularly when acoustic and RF signals are combined [6, 12]; accuracies of a few percent of the transmission range are reported. Acoustic signals, however, are temperature dependent and require an unobstructed line of sight. Furthermore, even small errors do accumulate when propagating distance information over multiple hops.

A drastic approach that avoids the range error problem altogether is to use only connectivity between nodes. The GPS-less system by Bulusu et al. [3] employs a grid of *beacon* nodes with known locations; each unknown node sets its position to the centroid of the locations of the beacons connected to the unknown. The position accuracy is about one-third of the separation distance between beacons, implying a high beacon density for practical purposes. Doherty et al. use the connectivity between nodes to formulate a set of geometric constraints and solve it using convex optimization [5]. The resulting accuracy depends on the fraction of anchor nodes. For example, with 10% anchors the accuracy for unknowns is on the order of the radio range. A serious drawback, which is currently being addressed, is that convex optimization is performed by a single, centralized node. The "DV-hop" approach by Niculescu and Nath, in contrast, is completely ad-hoc and achieves an accuracy of about one-third of the radio range for networks with dense populations of (highly connected) nodes [10]. In a first phase anchors flood their location to all nodes in the network. Each unknown node records the position and (minimum) number of hops to at least three anchors. Whenever an anchor  $a_1$  infers the position of another anchor  $a_2$  it computes the distance between them, divides that by the number of hops, and floods this average hop distance into the network. Each unknown uses the average hop distance to convert hop counts to distances, and then performs a triangulation to three or more distant anchors to estimate its own position. "DV-hop" works well in dense and regular topologies, but for sparse or irregular networks the accuracy degrades to the radio range.



More accurate positions can be obtained by using the range measurements between individual nodes (when the errors are small). When the fraction of anchor nodes is high the “iterative multilateration” method by Savvides et al. can be used [12]. Nodes that are connected to at least three anchors compute their position and upgrade to anchor status, allowing additional unknowns to compute their position in the next iteration, etc. Recently a number of approaches have been proposed that require few anchors [4, 9, 10, 11]. They are quite similar and operate as follows. A node measures the distances to its neighbors and then broadcasts this information. This results in each node knowing the distance to its neighbors *and* some distances between those neighbors. This allows for the construction of (partial) local maps with relative positions. Adjacent local maps are combined by aligning (mirroring, rotating) the coordinate systems. The known positions of the anchor nodes are used to obtain maps with absolute positions. When three or more anchors are present in the network a single absolute map results. This style of locationing is not very robust since range errors accumulate when combining the maps.

### 3 Two-phase positioning

As mentioned earlier, the two primary obstacles to positioning in an ad-hoc network are the sparse anchor node problem and the range error problem. In order to address each of these problems sufficiently, our algorithm is separated into two phases: start-up and refinement. For the start-up phase we use Hop-TERRAIN, an in-house algorithm similar to DV-hop [10]. The Hop-TERRAIN algorithm is run once at the beginning of the positioning algorithm to overcome the sparse anchor node problem, and the Refinement algorithm is run iteratively afterwards to improve upon and refine the position estimates generated by Hop-TERRAIN. Note therefore that the emphasis for Hop-TERRAIN is not on getting highly accurate position estimates, but instead on getting very rough estimates so as to have a starting point for Refinement. Conversely, Refinement is concerned only with nodes that exist within a one-hop neighborhood, and it focuses on increasing the accuracy of the position estimates as much as possible.

#### 3.1 Hop-TERRAIN

Before the positioning algorithm has started, most of the nodes in a network have no positioning data, with

the exception of the anchors. The networks being considered for this algorithm will be scalable to very large numbers of nodes spread over large areas, relative to the short radio ranges that each of the nodes is expected to possess. Furthermore, it is expected that the percentage of nodes that are anchor nodes will be small. This results in a situation in which only a very small percentage of the nodes in the network are able to establish direct contact with any of the anchors, and probably none of the nodes in the network will be able to directly contact enough anchors to derive a position estimate.

In order to overcome this initial information deficiency, the Hop-TERRAIN algorithm finds the number of hops from a node to each of the anchor nodes in a network and then multiplies this hop count by an average hop distance (see Section 4.2) to estimate the range between the node and each anchor. These computed ranges are then used together with the anchor nodes’ known positions to perform a triangulation and get the node’s estimated position. The triangulation consists of solving a system of linearized equations ( $Ax=b$ ) by means of a least squares algorithm, as in earlier work [11].

Each of the anchor nodes launches the Hop-TERRAIN algorithm by initiating a broadcast containing its known location and a hop count of 0. All of the one-hop neighbors surrounding an anchor hear this broadcast, record the anchor’s position and a hop count of 1, and then perform another broadcast containing the anchor’s position and a hop count of 1. Every node that hears this broadcast and did not hear the previous broadcasts will record the anchor’s position and a hop count of 2 and then rebroadcast. This process continues until each anchor’s position and an associated hop count value have been spread to every node in the network. It is important that nodes receiving these broadcasts search for the smallest number of hops to each anchor. This ensures conformity with the model used to estimate the average distance of a hop, and it also greatly reduces network traffic.

As broadcasts may be omni-directional, and may therefore reach nodes behind the broadcasting node (relative to the direction of the flow of information), this algorithm causes nodes to hear many more packets than necessary. In order to prevent an infinite loop of broadcasts, nodes are allowed to broadcast information only if it is not stale to them. In this context, information is stale if it refers to an anchor that the node has already heard from and if the hop count included in the arriving packet is greater than or equal to the hop count stored in memory for this particular anchor. New information will always trigger a broadcast, whereas stale information will never trigger a broadcast.

Once a node has received an average hop distance and data regarding at least 3(4) anchor nodes for a network existing in a 2(3)-dimensional space, it is able to perform a triangulation to estimate its location. If this node subsequently receives new data after already having performed a triangulation, either a smaller hop count or a new anchor, the node simply performs another triangulation to include the new data. This procedure is summarized in the following piece of pseudo code:

```

when a positioning packet is received,
  if new anchor or lower hop count
  then
    store hop count for this anchor.
    broadcast new packet for this anchor with
    hop count = (hop count + 1).
  else
    do nothing.
  if average hop count is known and
    number of anchors  $\geq$  (dimension of space + 1)
  then
    triangulate.
  else
    do nothing.

```

The resulting position estimate is likely to be coarse in terms of accuracy, but it provides an initial condition from which Refinement can launch. The performance of this algorithm is discussed in detail in Section 5.

### 3.2 Refinement

Given the initial position estimates of Hop-TERRAIN in the start-up phase, the objective of the refinement phase is to obtain more accurate positions using the estimated ranges between nodes. Since Refinement must operate in an ad-hoc network, only the distances to the direct (one-hop) neighbors of a node are considered. This limitation allows Refinement to scale to arbitrary network sizes and to operate on low-level networks that do not support multi-hop routing (only a local broadcast is required).

Refinement is an iterative algorithm in which the nodes update their positions in a number of steps. At the beginning of each step a node broadcasts its position estimate, receives the positions and corresponding range estimates from its neighbors, and computes a least squares triangulation solution to determine its new position. In many cases the constraints imposed by the distances to the neighboring locations will force the new position towards the true position of the node. When,

after a number of iterations, the position update becomes small Refinement stops and reports the final position. Note that Refinement is by nature an ad-hoc (distributed) algorithm.

The beauty of Refinement is its simplicity, but that also limits its applicability. In particular, it was *a priori* not clear under what conditions Refinement would converge and how accurate the final solution would be. A number of factors that influence the convergence and accuracy of iterative Refinement are:

- the accuracy of the initial position estimates,
- the magnitude of errors in the range estimates,
- the average number of neighbors, and
- the fraction of anchor nodes.

Based on previous experience we assume that redundancy can counter the above influences to a large extent. When a node has more than 3(4) neighbors in a 2(3)-dimensional space the induced system of linear equations is over-defined and errors will be averaged out by the least squares solver. For example, data collected by Beutel [1] shows that large range errors (standard deviation of 50%) can be tolerated when locating a node surrounded by 5 (or more) anchors in a 2-dimensional space: the average distance between the estimated and true position of the node is less than 5% of the radio range.

Despite the positive effects from redundancy we observed that a straightforward application of Refinement did not converge in a considerable number of “reasonable” cases. Close inspection of the sequence of steps taken under Refinement revealed two important causes:

1. Errors propagate fast throughout the whole network. If the network has a diameter  $d$ , then an error introduced by a node in step  $s$  has (indirectly) affected every node in the network by step  $s + d$  because of the triangulate-hop-triangulate-hop... pattern.
2. Some network topologies are inherently hard, or even impossible, to locate. For example, a cluster of  $n$  nodes (no anchors) connected by a single link to the main network can be simply rotated around the ‘entry’-point into the network while keeping the exact same intra- node ranges. Another example is given in Figure 1.

To mitigate error propagation we modified the refinement algorithm to include a confidence associated with each node’s position. The confidences are used to weigh the equations when solving the system of linear equations. Instead of solving  $Ax=b$  we now solve

$wAx=wb$ , where  $w$  is the vector of confidence weights. Nodes, like anchors, that have high faith in their position estimates select high confidence values (close to 1). A node that observes poor conditions (e.g., few neighbors, poor constellation) associates a low confidence (close to 0) with its position estimate, and consequently has less impact on the outcome of the triangulations performed by its neighbors. The details of confidence selection will be discussed in Section 4.3. The usage of confidence weights improved the behavior of Refinement greatly: almost all cases converge now, and the accuracy of the positions is also improved considerably.

Another improvement to Refinement was necessary to handle the second issue of ill-connected groups of nodes. Detecting that a single node is ill-connected is easy: if the number of neighbors is less than 3(4) then the node is ill-connected in a 2(3)-dimensional space. Detecting that a group of nodes is ill-connected, however, is more complicated since some global overview is necessary. We employ a heuristic that operates in an ad-hoc fashion (no centralized computation), yet is able to detect most ill-connected nodes. The underlying premise for the heuristic is that a sound node has *independent* references to at least 3(4) anchors. That is, the multi-hop routes to the anchors have no link (edge) in common. For example, node 3 in Figure 1 (which is taken from [12]) meets this criteria and is considered sound.

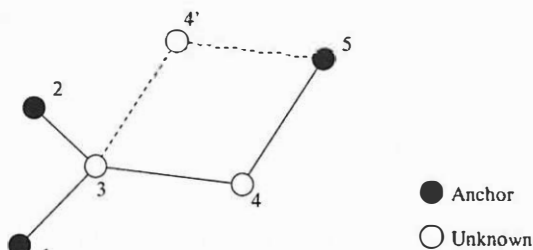


Figure 1: Example topology.

To determine if a node is sound, the Hop-TERRAIN algorithm records the ID of each node's immediate neighbor along a shortest path to each anchor. When multiple shortest paths are available, the first one discovered is used (this only approximates the intended condition but is considerably simpler). These IDs are collected in a set of sound neighbors. When the number of unique IDs in this set reaches 3(4), a node declares itself sound and may enter the Refinement phase. The neighbors of the sound node add its ID to their sets and may in turn become sound if their sound sets become sufficient. This process continues throughout the network. The end result is that most ill-connected nodes will not be able to fill their sets of sound neighbors with enough entries and,

therefore, may not participate in the Refinement phase. In the example topology in Figure 1, node 3 will become sound, but node 4 will not. We also note that the more restrictive *participating node* definition by Savvides et al. renders both unknown nodes as ill-conditioned [12].

Refinement with both modifications (confidence weights, detection of ill-connected nodes) performs quite satisfactorily, as will be shown by the experiments in Section 5.

## 4 Simulation and algorithm details

To study the robustness of our two-phase positioning algorithm we created a simulation environment in which we can easily control a number of (network) parameters. We implemented the Hop-TERRAIN and Refinement algorithms as C++ code running under the control of the OMNeT++ discrete event simulator [13]. The algorithms are event driven, where an event can be an incoming message or a periodic timer. Processing an event usually involves updating internal state, and often generates output messages that must be broadcast. All simulated sensor nodes run exactly the same C++ code. The OMNeT++ library is in control of the simulated time and enforces a semi-concurrent execution of the code 'running' on the multiple sensor nodes.

### 4.1 Network layer

Although our positioning algorithm is designed to be used in an ad-hoc network that presumably employs multi-hop routing algorithms, our algorithm only requires that a node be able to broadcast a message to all of its one hop neighbors. An important result of this is the ability for system designers to allow the routing protocols to rely on position information, rather than the positioning algorithm relying on routing capabilities.

An important issue is whether or not the network provides reliable communication in the presence of concurrent transmission. In this paper we assume that message loss or corruption does not occur and that each message is delivered at the neighbors within a fixed radio range ( $R$ ) from the sending node. Concurrent transmissions are allowed when the transmission areas (circles) do not overlap. A node wanting to broadcast a message while another message in its area is in progress must wait until that transmission (and possibly other queued messages) completes. In effect we employ a CSMA policy.

The functionality of the network layer (local broadcast) is implemented in a single OMNeT++ object, which is connected to all sensor-node objects in the simulation. This network object holds the topology of the simulated sensor network, which can be read from a "scenario" file or generated at random at initialization time. At time zero the network object sends a pseudo message to each sensor-node object telling its role (anchor or unknown) and some attributes (e.g., the position in the case of an anchor node). From then on it relays messages generated by sensor nodes to the sender's neighbors within a radius of  $R$  units.

## 4.2 Hop-TERRAIN

At time zero of the Hop-TERRAIN algorithm, all of the nodes in the network are waiting to receive hop count packets informing them of the positions and hop distances associated with each of the anchor nodes. Also at time zero, each of the anchor nodes in the network broadcasts a hop count packet, which is received and repeated by all of the anchors' one-hop neighbors. This information is propagated throughout the network until, ideally, all the nodes in the network have positions and hop counts for all of the anchors in the network as well as an average hop distance (see below). At this point, each of the nodes performs a triangulation to create an initial estimate of its position. The number of anchors in any particular scenario is not known by the nodes in the network, however, so it is difficult to define a stopping criteria to dictate when a node should stop waiting for more information before performing a triangulation. To solve this problem, nodes perform triangulations every time they receive information that is not stale after having received information from the first 3(4) anchors in a 2(3)-dimensional space (see Section 3.1 for a definition of stale information).

Nodes also rely on the anchor nodes to inform them of the value to use for the assumed average hop distance used in calculating the estimated range to each anchor. Initially we experimented with simply using the maximum radio range for this quantity. Better position results, however, are attained by dynamically determining the average hop distance by comparing the number of hops between the anchors themselves to the known distances separating them following the calibration procedure used for DV-hop (see Section 2). We implemented the calibration procedure as a separate pass that follows the initial hop-count flooding. When an anchor node receives a hop count from another anchor it computes its estimate of the average hop distance, and floods that back into the network. Nodes wait for

the first such estimate to arrive before performing any triangulation as outlined above. Subsequent estimates from other anchor pairs are simply discarded to reduce network load.

The above details are sufficient for controlling the Hop-TERRAIN algorithm within a simulated environment where all of the nodes start up at the same time. One important consequence of a real network, however, is that the nodes in the network start up or enter the network at random times, relative to each other. This allows for the possibility that a late node might miss some of the waves of propagated broadcast messages originating at the anchor nodes. To solve this, each node is programmed to announce itself when it first comes online in a new network. Likewise, every node is programmed to respond to these announcements by passing the new node their own position estimates, the positions of all of the anchor nodes they know of, and the hop counts and hop distance metrics associated with these anchors. Note that, according to the rebroadcast rules regarding stale information, this information will all be new to the new node, causing this new node to then rebroadcast all of the information to all of its one-hop neighbors. This becomes important in the cases where the new node forms a link between two clusters of nodes that were previously not connected. In cases where all or most of the new node's one-hop neighbors came online before the new node, this information will most likely be considered stale, and so these broadcasts will not be repeated past a distance of one hop.

## 4.3 Refinement

The refinement algorithm is implemented as a periodic process. The information in incoming messages is recorded internally, but not processed immediately. This allows for accumulating multiple position updates from different neighbors, and responding with a single reply (outgoing broadcast message). The task of an anchor node is very simple: it broadcasts its position whenever it has detected a new neighbor in the preceding period. The task of an unknown node is more complicated. If new information arrived in the preceding period it performs a triangulation to compute a new position estimate, determines an associated confidence level, and finally decides whether or not to send out a position update to its neighbors.

A confidence is a value between 0 and 1. Anchors immediately start off with confidence 1; unknown nodes start off at a low value (0.1) and may raise their confidence at subsequent Refinement iterations. Whenever a node

performs a successful triangulation it sets its confidence to the average of its neighbors' confidences. This will, in general, raise the confidence level. Nodes close to anchors will raise their confidence at the first triangulation, raising in turn the confidence of nodes two hops away from anchors on the next iteration, etc. Triangulations sometimes fail or the new position is rejected on other grounds (see below). In these cases the confidence is set to zero, so neighbors will not be using erroneous information of the inconsistent node in the next iteration. This generally leads to new neighbor positions bringing the faulty node back into a consistent state, allowing it to build its confidence level again. In unfortunate cases a node keeps getting back into an inconsistent state, never converging to a final position/confidence. To warrant termination we simply limit the number of position updates of a node to a maximum. Nodes that end up with a poor confidence ( $< 0.1$ ) are discarded and excluded from the reported error results; all others are considered to be located and included in the results.

To avoid flooding the network with insignificant or erroneous position updates the triangulation results are classified as follows. First, a triangulation may simply fail because the system of equations is underdetermined (too few neighbors, bad constellation). Second, the new position may be very close to the current one, rendering the position update insignificant. We use a tight cut-off radius of  $\frac{1}{10.00}$  of the radio range; experimentation showed Refinement is fairly insensitive to this value as long as it is small (under 1% of the radio range). Third, we check that the new position is within the reach of the anchors used by Hop-TERRAIN. Similarly to Doherty et al. [5] we check the convex constraints that the distance between the position estimate and anchor  $a_i$  must be less than the length of the shortest path to  $a_i$  (hop-count $_i$ ) times the radio range ( $R$ ). When the position drifts outside the convex region, we reset the position to the original initial position computed by Hop-TERRAIN. Finally, the validity of the new position is checked by computing the difference between the sum of the observed ranges and the sum of the distances between the new position and the neighbor locations. Dividing this difference by the number of neighbors yields a normalized residue. If the residue is large (residue  $>$  radio range) we assume that the system of equations is inconsistent and reject the new position. To avoid being trapped in some local minima, however, we occasionally accept bad moves (10% chance), similar to a simulated annealing procedure (without cooling down), and reduce the confidence by 50%.

An unexpected source of errors is that Hop-TERRAIN assigns the same initial position to all nodes with identical hop counts to the anchors. For example, twin

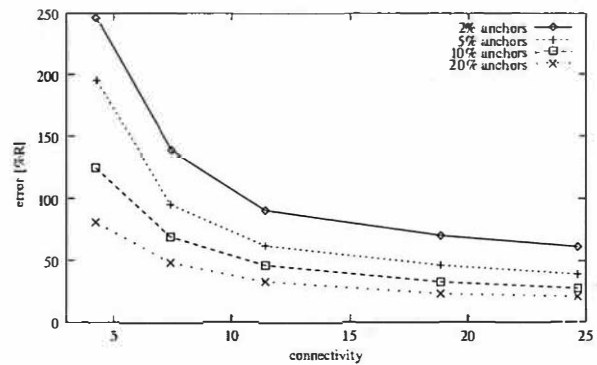


Figure 2: Average position error after Hop-TERRAIN (5% range errors).

nodes that share the exact same set of neighbors are both assigned the same initial position. The consequence is that a neighbor of two 'look-alikes' is confronted with a large inconsistency: two nodes that share the same position have two different range estimates. Simply dropping one of the two equations from the triangulation yields better position estimates in the first iteration of Refinement and even has a noticeable impact on the accuracy of the final position estimates.

## 5 Experiments

In order to evaluate our algorithm, we ran many experiments on both Hop-TERRAIN and Refinement using the OMNeT++ simulation environment. All data points represent averages over 100 trials in networks containing 400 nodes. The nodes are randomly placed, with a uniform distribution, within a square area. The specified fraction of anchors is randomly selected, and the range between connected nodes is blurred by drawing a random value from a normal distribution having a parameterized standard deviation and having the true range as the mean<sup>1</sup>. The connectivity (average number of neighbors) is controlled by specifying the radio range. To allow for easy comparison between different scenarios, range errors as well as errors on position estimates are normalized to the radio range (i.e. 50% position error means half the range of the radio).

Figure 2 shows the average performance of the Hop-TERRAIN algorithm as a function of connectivity and anchor population in the presence of 5% range errors. As seen in this plot, position estimates by Hop-TERRAIN

<sup>1</sup> Ranges are enforced to be non-negative by clipping values below zero.



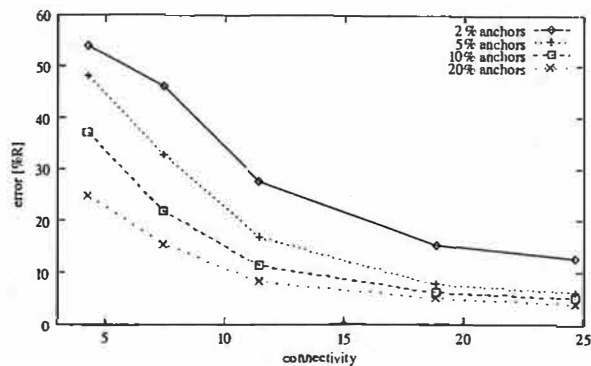


Figure 3: Average position error after Refinement (5% range errors).

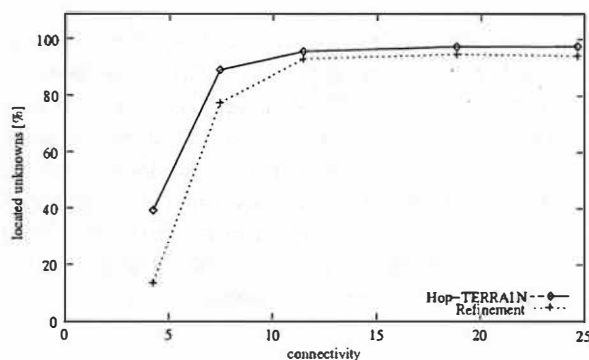


Figure 4: Fraction of located nodes (2% anchors, 5% range errors).

have an average accuracy under 100% error in scenarios with at least 5% anchor population and an average connectivity level of 7 or greater. In extreme situations where very few anchors exist and connectivity in the network is very low, Hop-TERRAIN errors reach above 250%.

Figure 3 displays the results from the same experiment depicted in Figure 2, but now the position estimates of Hop-TERRAIN are subsequently processed by the Refinement algorithm. Its shape is similar to that of Figure 2, showing relatively consistent error levels of less than 33% in scenarios with at least 5% anchor population and an average connectivity level of 7 or greater. Refinement also has problems with low connectivity and anchor populations, and is shown to climb above 50% position error in these harsh conditions. Overall Refinement improves the accuracy of the position estimates by Hop-TERRAIN by a factor three to five.

Figure 4 helps to explain the sharp increases in positioning errors for low anchor populations and sparse

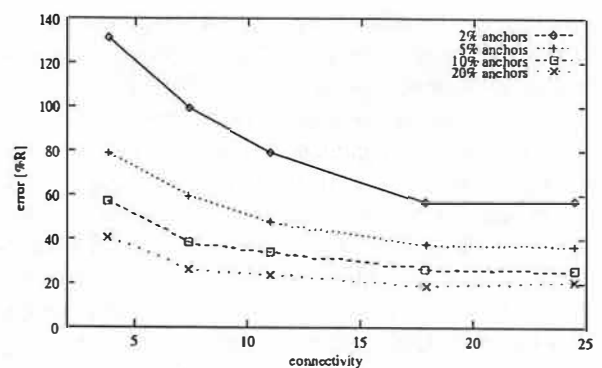


Figure 5: Average position error after Hop-TERRAIN (2D grid, 5% range errors).

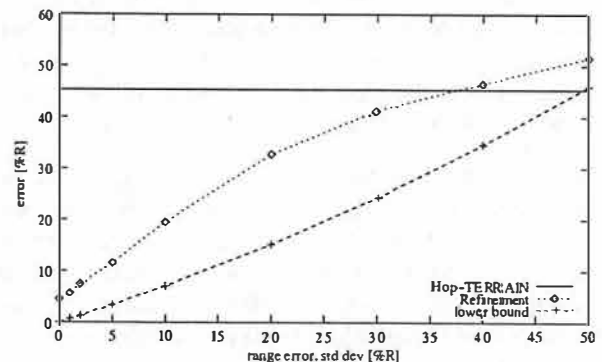


Figure 6: Range error sensitivity (10% anchors, connectivity 12).

networks shown in figures 2 and 3. Figure 4 shows that, as the average connectivity between nodes throughout the network decreases past certain points, both algorithms break down, failing to derive position estimates for large fractions of the network. This is due simply to a lacking of sufficient information, and is a necessary consequence of loosely connected networks. Nodes can only be located when connected to at least 3(4) neighbors; Refinement also requires a minimal confidence level (0.1). It should be noted that the results in Figure 4 imply that the reported average position errors for low connectivities in figures 2 and 3 have low statistical significance, as these points represent only small fractions of the total network. Nevertheless, the general conclusion to be drawn from figures 2, 3, and 4 is that both Hop-TERRAIN and Refinement perform poorly in networks with average connectivity levels of less than 7.

Since connectivity has a pronounced effect on position error we were interested if other topological characteristics would show large effects as well. In the following

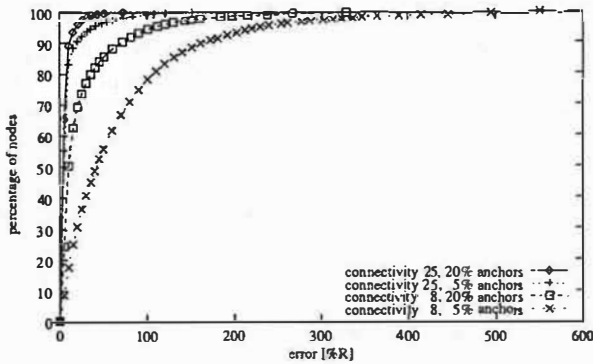


Figure 7: Cumulative error distribution (5% range errors).

experiment we randomly place 400 nodes on the vertices of a 200x200 grid, rather than allowing the nodes to sit anywhere in the square area. We found that the grid layout did not result in better performance for the Refinement algorithm, relative to the performance of the Refinement algorithm with random node placement. We do not include a plot here because it looks almost identical to Figure 3. We did find a difference in performance for Hop-TERRAIN though. Figure 5 shows that placing the nodes on a grid dramatically reduces the errors of the Hop-TERRAIN algorithm in the cases where connectivity or anchor node populations are low. For example, with 5% anchors and a connectivity of 8 nodes, the average position error decreases from 95% (random distribution) to 60% (grid). We suspect this is due to the consistent distances between nodes, the ideal topologies within clusters that result form the grid layout, and the inherently optimized connectivity levels across the entire network.

Sensitivity to average error levels in the range measurements is a major concern for positioning algorithms. Figure 6 shows the results of an experiment in which we held anchor population and connectivity constant at 10% and 12 nodes, respectively, while varying the average level of error in the range measurements. We found that Hop-TERRAIN was almost completely insensitive to range errors. This is a result of the binary nature of the procedure in which routing hops are counted; if nodes can see each other, they pass on incremented hop counts, but at no time do any nodes attempt to measure the actual ranges between them. Unlike Hop-TERRAIN, Refinement does rely on the range measurements performed between nodes, and Figure 6 shows this dependence accordingly. At less than 40% error in the range measurements, on average, Refinement offers improved position estimates over Hop-TERRAIN. The results improve steadily as the range errors decrease.

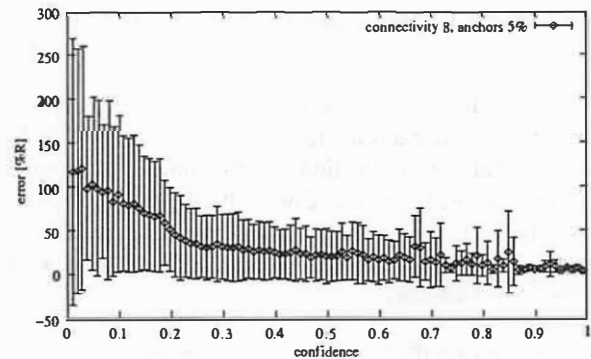
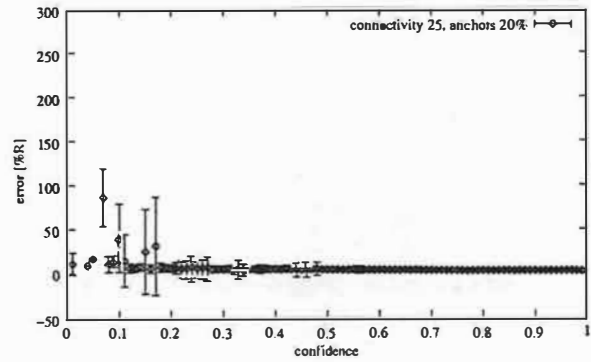


Figure 8: Relation between confidence and positioning error (average and standard deviation).

For reference we determined the best possible position information that can be obtained in each case. For each node we performed a triangulation using the *true* positions of its neighbors and the corresponding erroneous range measurements. The resulting position errors are plotted as the lower bound in Figure 6. This suggests that there is room for improvement for Refinement.

Up until this point we reported average position errors. Figure 7, in contrast, gives a detailed look at the distribution of the position errors for individual nodes under four different scenarios. Note that the distributions have similar shapes: many nodes with small errors, large tails with outliers. Refinement's confidence metrics are to some extent capable of pinpointing the outliers. Figure 8 shows the relationship between position error levels and the corresponding confidence values assigned to each node. The data for Figure 8 was taken from the best and worst case scenarios from the same experiment used to generate Figure 7. As desired, the nodes with higher position errors are assigned lower confidence levels. In the easier case, the confidence indicators are much more reliable than in the more difficult case. The large standard deviations, however, show that confidence is

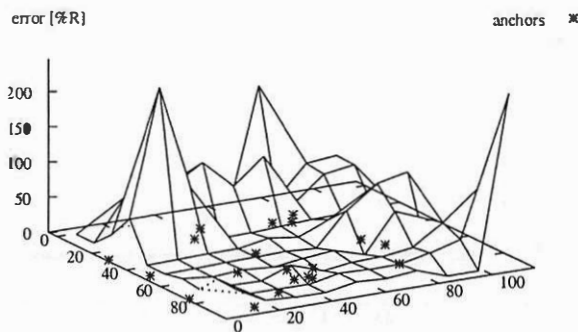


Figure 9: Geographic error distribution (5% anchors, connectivity 12, 5% range errors).

not a good indicator for position accuracy. This is unfortunate since a reliable confidence metric would be very useful for applications, for example, to identify regions of “bad” nodes. Currently, the value of using confidence levels is the improved average positioning error compared to a naive implementation of Refinement without confidences.

Finally, yet another useful way of looking at the distribution of errors over individual nodes is to take their geographical location into account. Figure 9 plots positioning errors as a function of a node’s location in the square testing area. This experiment used 400 randomly placed nodes, an anchor population of 5%, an average connectivity level of 12, and range errors of 5%. The error distribution in Figure 9 is quite typical for many scenarios showing that areas along the edges of the network lacking a high concentration of anchor nodes are particularly susceptible to high position errors.

## 6 Discussion

It is interesting to compare our results from the previous section with the alternative approaches discussed in Section 2. First, we discuss the performance of Hop-TERRAIN and related algorithms that do not use range measurements. Hop-TERRAIN is similar to the “DV-hop” algorithm by Niculescu and Nath [10], but we get consistently higher position errors, for example, 69% (Hop-TERRAIN) versus 35% (DV-hop) on a scenario with 10% anchors and a connectivity of 8. Under poorer network conditions though, Hop-TERRAIN is more robust than DV-hop, showing about a factor of 2 improvement in position accuracy in sparsely connected networks. Regardless, the trend observed in both studies is the same: when the fraction of anchors

drops below 5%, position errors rapidly increase. The convex optimization technique by Doherty et al. [5] is about as accurate as Hop-TERRAIN, except for very low fractions of anchors. For example, convex optimization achieves position errors that are above 150% on a scenario (200 nodes, 5% anchors, connectivity of 6) where Hop-TERRAIN errors are around 125%; the gap grows for even lower fractions of anchors. As mentioned earlier, convex optimization is a centralized algorithm.

The results of Refinement are comparable to those reported by Savvides et al. for an “iterative multilateration” scenario with 50 nodes, 20% anchors, connectivity 10, and 1% range errors [12]. Their algorithm, however, can handle neither low anchor fractions nor low connectivities, because positioning starts from nodes connected to at least 3 anchors. Refinement still performs acceptably well with few anchors or a low connectivity. Furthermore the preliminary results of their more advanced “collaborative multilateration” algorithm show that Refinement is able to determine the position of a larger fraction of unknowns: 56% (Refinement) versus 10% (collaborative multilateration) on a scenario with just 5% anchors (200 nodes, connectivity 6).

The “Euclidean” algorithm by Niculescu and Nath uses range estimates to construct local maps that are unified into a single global map [10]. The results reported for random configurations show that “Euclidean” is rather sensitive to range errors, especially with low fractions of anchors: in case of 10% anchors their Hop-TERRAIN equivalent (DV-hop) outperforms Euclidean. Refinement achieves better position estimates *and* is more robust since the cross over with Hop-TERRAIN occurs around 40% range errors (see Figure 6).

In summary, the performance of Hop-TERRAIN and Refinement is comparable to other algorithms in the case of “easy” network topologies (high connectivity, many anchors) with low range errors, and outperforms the competition in difficult cases (low connectivity, few anchors, large range errors). The results of refinement can most likely be improved even further when the placement of anchors nodes can be controlled given the positive experience reported by others [2, 5]. Since the largest errors occur along the edges of the network (see Figure 9), most anchors should be placed on the perimeter of the network. Another approach to increase the accuracy of locationing systems is to use other sources of information. When locating sensors in a room, for example, knowing that the sensors are wall mounted eliminates one degree of freedom. Incorporating such knowledge in localization algorithms, however, requires great care. For example, knowing that two sensors cannot communicate does not imply that they

are located far apart since a wall may simply prohibit radio communication.

Based on the experimental results from Section 5 and the discussion above we recommend a number of guidelines for the installation of wireless sensor networks:

- place anchors carefully (i.e. at the edges), and either
- ensure a high connectivity ( $> 10$ ), or
- employ a reasonable fraction of anchors ( $> 5\%$ ).

This will create the best conditions for positioning algorithms in general, and for Hop-TERRAIN and Refinement in particular.

## 7 Conclusions and future work

In this paper we have presented a completely distributed algorithm for solving the problem of positioning nodes within an ad-hoc, wireless network of sensor nodes. The procedure is partitioned into two algorithms: Hop-TERRAIN and Refinement. Each algorithm is described in detail. The simulation environment used to evaluate these algorithms is explained, including details about the specific implementation of each algorithm. Many experiments are documented for each algorithm, showing several aspects of the performance achieved under many different scenarios. The results show that we are able to achieve position errors of less than 33% in a scenario with 5% range measurement error, 5% anchor population, and an average connectivity of 7 nodes. Finally, guidelines for implementing and deploying a network that will use these algorithms are given and explained.

An important aspect of wireless sensor networks is energy consumption. In the near future we therefore plan to study the amount of communication and computation induced by running Hop-TERRAIN and Refinement. A particularly interesting aspect is how the accuracy vs. energy consumption trade-off changes over subsequent iterations of Refinement.

## Acknowledgements

We would like to thank DARPA for funding the Berkeley Wireless Research Center under the PAC-C program (Grant #F29601-99-1-0169). Also, Koen Langendoen was supported by the USENIX Research Exchange (ReX) program, which allowed him to visit the BWRC

for the summer of 2001 and work on the Refinement algorithm. Finally, we would like to thank the anonymous reviewers and our “shepherd” Mike Spreitzer for their constructive comments on the draft version of this paper.

## References

- [1] J. Beutel. Geolocation in a Pico Radio environment. Master's thesis, ETH Zürich. December 1999.
- [2] N. Bulusu, J. Heidemann, V. Bychkovskiy, and D. Estrin. Density-adaptive beacon placement algorithms for localization in ad hoc wireless networks. In *IEEE Infocom 2002*, New York, NY, June 2002.
- [3] N. Bulusu, J. Heidemann, and D. Estrin. GPS-less low-cost outdoor localization for very small devices. *IEEE Personal Communications*, 7(5):28–34, Oct 2000.
- [4] S. Capkun, M. Hamdi, and J.-P. Hubaux. GPS-free positioning in mobile ad-hoc networks. In *Hawaii Int. Conf. on System Sciences (HICSS-34)*, pages 3481–3490, Maui, Hawaii, January 2001.
- [5] L. Doherty, K. Pister, and L. El Ghaoui. Convex position estimation in wireless sensor networks. In *IEEE Infocom 2001*, Anchorage, AK, April 2001.
- [6] L. Girod and D. Estrin. Robust range estimation using acoustic and multimodal sensing. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, Maui, Hawaii, October 2001.
- [7] J. Hightower and G. Borriello. Location systems for ubiquitous computing. *IEEE Computer*, 34(8):57–66, Aug 2001.
- [8] J. Hightower, R. Want, and G. Borriello. SpotON: An indoor 3D location sensing technology based on RF signal strength. UW CSE 00-02-02, University of Washington, Department of Computer Science and Engineering, Seattle, WA, February 2000.
- [9] R. Mukai, R. Hudson, G. Pottie, and K. Yao. A protocol for distributed node location. *IEEE Communication Letters*, to be published.
- [10] D. Niculescu and B. Nath. Ad-hoc positioning system. In *IEEE GlobeCom*, November 2001.
- [11] C. Savarese, J. Rabaey, and J. Beutel. Locationing in distributed ad-hoc wireless sensor networks. In *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 2037–2040, Salt Lake City, UT, May 2001.
- [12] A. Savvides, C.-C. Han, and M. Srivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *7th ACM Int. Conf. on Mobile Computing and Networking (Mobicom)*, pages 166–179, Rome, Italy, July 2001.
- [13] A. Varga. The OMNeT++ discrete event simulation system. In *European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic, June 2001.





# Application-specific network management for energy-aware streaming of popular multimedia formats

Surendar Chandra  
University of Georgia  
Athens, GA 30602-7404  
surendar@cs.uga.edu

Amin Vahdat  
Duke University  
Durham, NC 27709-0129  
vahdat@cs.duke.edu

## Abstract

The typical duration of multimedia streams makes wireless network interface (WNIC) energy consumption a particularly acute problem for mobile clients. In this work, we explore ways to transmit data packets in a predictable fashion; allowing the clients to transition the WNIC to a lower power consuming *sleep* state. First, we show the limitations of IEEE 802.11 power saving mode for isochronous multimedia streams. Without an understanding of the stream requirements, they do not offer any energy savings for multimedia streams over 56 kbps. The potential energy savings is also affected by multiple clients sharing the same access point. On the other hand, an application-specific server side traffic shaping mechanism can offer good energy saving for all the stream formats without any data loss. We show that the mechanism can save up to 83% of the energy required for receiving data. The technique offers similar savings for multiple clients sharing the same wireless access point. For high fidelity streams, media players react to these added delays by lowering the stream fidelity. We propose that future media players should offer configurable settings for recognizing such energy-aware packet delay mechanisms.

## 1 Introduction

The proliferation of inexpensive, multimedia capable mobile devices and ubiquitous high-speed network technologies to deliver multimedia objects is fueling the demand for mobile streaming multimedia. Public venues [31] are deploying high speed IEEE 802.11b [24] based public wireless LAN networks. Commodity PDA devices that allow the users to consume mobile streaming multimedia are becoming popular. A necessary feature for mass acceptance of a streaming multimedia device

is acceptable battery life. Advances in hardware and software technologies have not been matched by corresponding improvements in battery technologies. Future trends in battery technologies alone (along with the continual pressure for further device miniaturization) do not promise dramatic improvements that will make this issue disappear.

Newer hardware improvements are reducing the power consumption of system components such as back-lit displays, CPUs etc. However, WNICs operating at the same frequency band and range continue to consume significant power. Earlier work by Stemm et al. [32] reported that the network interfaces draw significant amounts of power. For example, a 2.4 GHz Wavelan DSSS card (11 Mbps) consumes 177 mW while in *sleep* state, but consumes 1319 mW while *idle*. Havinga et al. [18] noted that this Wavelan card consumes 1425 mW for receiving data and 1675 mW for transmitting data. For comparison, a fully operational Compaq iPAQ PDA only consumes 929 mW while the same iPAQ consumes 470 mW with the backlight turned off [14, 8]. For reference, the iPAQ is equipped with two 2850 mWh batteries, one each in the unit and the PCMCIA sleeve. Streaming media tends to be large and long running and consume significant amounts of network resources to download data. Hence, it is important to look at techniques to reduce the energy consumed by the network interface to download the multimedia stream.

Traditionally, reducing the fidelity of the stream and hence the size is a popular technique that is used to customize the multimedia stream for a low bandwidth network. Reducing multimedia fidelity can also be expected to reduce the amount of data and hence the total energy consumed. However, if care is not taken to return the network interface to the *sleep* state as much as possible, reducing the amount of transmitted data will have negligible effect on the overall client energy consumption. Frequent switching to low power consump-

tion states also promises the added benefit of allowing the batteries to recover, exploiting the battery recovery effect [7].

In our earlier work [5], we explored the client WNIC energy implications of popular streaming formats (Microsoft media [28], Real media [29] and Quicktime [2]) under varying network conditions. We believe that these widely popular formats are more likely to be deployed than custom streaming formats (that are specially optimized for lower energy consumption). Based on our observations, we developed history based client-side techniques to exploit the stream behavior and lower the energy required to receive these streams. We illustrated the limitations of such client-side policies in predicting the next packet arrival times. These client-only policies do not allow us to achieve the potential energy saving for consuming multimedia streams without losing data packets.

In general, mechanisms that make the data packets arrive at predictable intervals can facilitate such transitions to lower power states. The choice of these transmission periods is a trade-off between frequent transitions to high power states and added delays in the multimedia stream reception. Such traffic shaping can be realized either in the origin server, in the network infrastructure closer to the mobile client and in the access point itself; at the MAC level. In this work, we analyze the effectiveness of these different approaches in regulating the streams to transmit data packets at regular, predictable intervals. Such packet arrivals enable client-side mechanisms to effectively transition the wireless interfaces to a lower power consuming *sleep* state.

In this work, we show the limitations of MAC level IEEE 802.11 power saving mode for isochronous multimedia streams. The access points have to balance potential energy savings for a single mobile client with a need for fair allocation of network resources. Without an understanding of the stream requirements, these MAC level mechanisms do not offer any energy savings for multimedia streams over 56 kbps. The potential energy savings also reduces for multiple clients sharing the same access point. On the other hand, a server side traffic shaping mechanism can offer good energy saving for all the stream formats without any data loss. We show that the mechanism can save up to 83% of the energy required for receiving useful data. The technique can also offer similar savings for multiple clients sharing the same wireless access point. For high fidelity streams, typical media systems react to these added delays by lowering the stream quality. We propose that future media players offer configurable settings for clients operat-

ing under energy conserving WLAN systems such that these delays are not associated with network congestion.

The remainder of this paper is organized as follows: Section 2 reviews our previous work as the necessary background and places our work in context to other related work. Next we present the experimental setup, evaluation methodologies, measurement metrics and the workloads used in our study in Section 3. Section 4 analyzes the effectiveness of IEEE 802.11 power management scheme to conserve energy for our streams. Section 5 explores the effectiveness of server side assistance in conserving the WNIC energy on the client. We conclude in Section 6.

## 2 Related work

There has been considerable work on power management for components of a mobile device. This work includes spindown policies for disks and alternatives [35, 3, 26, 12, 19], scheduling policies for reducing CPU energy consumption [34, 17] and managing wireless communications [20, 11, 21, 30]. Our work is similar in spirit to the work of Feeney et al. [15]. They obtain detailed measurements of the energy consumption of an IEEE 802.11 wireless network interface operating in an ad hoc networking environment. They showed that the energy consumption of an IEEE 802.11 wireless interface has a complex range of behavior and that the energy consumption was not synonymous with bandwidth utilization. Our work explores similar techniques for WNIC energy management for multimedia traffic.

Lorch et al. [27] presented a survey of the various software techniques for energy management. Havinga et al. [18] presented an overview of techniques for energy management of multimedia streams. Agrawal et al. [1] described techniques for processing video data for transmission under low battery power conditions. Corner et al. [10] described the time scales of adaptation for mobile wireless video-conferencing systems.

Ellis [13] advocates high level mechanisms for power management. Flinn et al. [16] demonstrated such a collaborative relationship between the operating system and application to meet user-specified goals for battery duration. Vahdat et al. [33] proposed that energy as a resource should be managed by the operating system. Kravets et al. [22] advocated an end-to-end model for conserving energy for wireless communications. In our earlier work [6], we utilized transcoding as an applica-

tion level technique to reduce the image data; trading off image size for network transmission and storage costs. In this work, we explore high level mechanisms to enable the mobile client to transition to lower power states and reduce overall energy requirements.

## 2.1 Client-side history based adaptation mechanism

In our earlier work [5], we explored the energy implications of popular streaming formats (Microsoft media [28], Real media [29] and Quicktime [2]) under varying network conditions. We believe that these widely popular formats are more likely to be deployed than custom streaming formats (that are specially optimized for lower energy consumption). We showed that Microsoft media tended to transmit packets at regular intervals. For high bandwidth streams, Microsoft media exploits network level fragmentation, which can lead to excessive packet loss (and wasted energy) in a lossy network. Real stream packets tend to be sent closer to each other, especially at higher bandwidths. Quicktime packets sometimes arrive in quick succession; most likely an application level fragmentation mechanism.

Based on our observations, we developed history based client-side techniques to exploit the stream behavior and lower the energy required to receive these streams. We illustrated the limitations of such client-side policies in predicting the next packet arrival times. We showed that the regularity of Microsoft media packet arrival rates allow simple, history based client-side policies to transition to lower power states with minimal data loss. A Microsoft media stream optimized for 28.8 Kbps can save over 80% in energy consumption with 2% data loss. A high bandwidth stream (768 Kbps) can still save 57% in energy consumption with less than 0.3% data loss. For comparison, the WNIC was only receiving data for 0.45% and 14.51% of the time for these two streams, respectively. Also, both Real and Quicktime packets were harder to predict at the client-side without understanding the semantics of the packets themselves. Quicktime's energy savings came at a high data loss, while Real offered negligible energy savings. We believe that modifying Real and Quicktime services to transmit larger data packets at regular intervals can offer better energy consumption characteristics with minimal latency and jitter.

## 2.2 MAC level power saving modes

Wireless technologies such as IEEE 802.11 [25, 24] and Bluetooth [4] use a scheduled rendezvous mechanism of power saving wherein the wireless nodes switch to a low power *sleep* mode and periodically awaken to receive data from other nodes. Different wireless technologies utilize variations of this scheduled rendezvous mechanism. For example, IEEE 802.11 uses scheduled beacons along with a TIM/DTIM packet notification mechanism. Bluetooth, a wireless cable replacement technology, defines three different power saving modes; the *sniff* mode which defines a variable slave specific activity delay interval, the *hold* mode wherein the slave can sleep for a predetermined interval without participating in the data traffic and the *park* mode wherein the slave gives up its active member address. The potential energy saving progressively decreases from *sniff* to *hold* to *park* modes. In general, Bluetooth enabled devices with their range less than 10 meters consume less power than IEEE 802.11 based WLAN devices. For example, a typical Bluetooth device (Ericsson PBA 313 01/2) consumes 84 mW to receive data, 126 mW to transmit data while consuming as little as 2 mW in the *park* state. Energy consumption in a fully packaged system is likely to be higher. Bluetooth technology offers a vertically integrated solution with a lower data rate and range as compared to wireless LAN technologies. As such, Bluetooth technologies are tuned towards cable replacement rather than for isochronous traffic generated by multimedia traffic.

## 3 System Architecture

In the last section we outlined earlier work on mechanisms for energy efficient multimedia streaming as well as the limitations of client-only policies in conserving energy without losing data packets. In this section, we describe the objectives, the system architecture and the experimental setup. We will highlight our experiences in the next two sections.

### 3.1 Objectives

Our primary goal was to reduce the energy required by the wireless client to consume a certain multimedia stream (of a given quality). We explore the effectiveness of energy aware traffic shaping in the network infrastructure closer to the mobile client and in the wireless

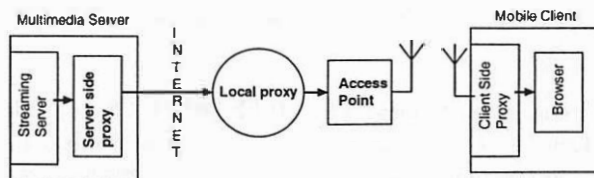


Figure 1: System Architecture

access point itself. The mechanism should also allow energy savings for multiple wireless clients sharing the same wireless access point.

Our experiments were designed to answer the following questions:

- Can we realize energy savings at the network MAC level without an understanding of the application level stream dynamics?
- Can traffic shaping assistance from the network infrastructure allow the client to transition to lower power consuming states more effectively?

### 3.2 Architecture

The system architecture is illustrated in Figure 1. The system consists of a server side proxy (SSP) or local proxy (LP) and a client-side proxy (CSP). The server side proxy allows the flexibility of traffic shaping at the source, without actually modifying the multimedia servers themselves. The local proxy performs similar functionality to a server side proxy and shares the same LAN network with the wireless client. The server side and local proxies can inform the client-side proxy of the next scheduled data burst. The client-side proxy interacts with the server side or local proxies. It is the responsibility of the client-side proxy to transition the WNIC to a lower power *sleep* state between scheduled data transfers. Since no data transfers are expected during this sleep interval, no data is lost.

### 3.3 Experiment Setup

The system setup that was used to customize the network transmissions to conserve energy for popular streaming formats is illustrated in Figure 2. The various components of our system are:

- **Multimedia Server:** Our multimedia server (Dell

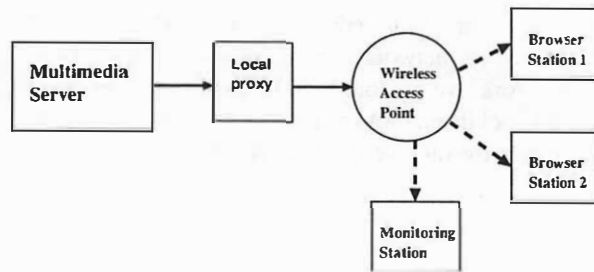


Figure 2: Experiment Setup

330) was equipped with a 1.5 GHz Pentium 4 with 512 MB of PC800 RDRAM memory, running Microsoft Windows 2000 Server SP2. The server was running Windows Media Service, Realserver 8.01 and Apple Darwin Server 3.0.1.

- **Wireless Access Point:** For our experiments, we used a dedicated D-Link DWL 1000, Orinoco RG 1000 and Orinoco AP 500 access points. The AP 500 was connected to an external range extender antenna. Throughout our experiments, we had turned off the security encryption feature of the access points.
- **Local proxy:** We used a Dell dual processor (Pentium Xeon 933 MHz) server with 1.5 GB of memory, running FreeBSD 4.4 (STABLE) for the local proxy. The proxy buffered packets from the multimedia servers and transmitted them after the configured delay. The proxy added these delays after transmitting all pending packets.
- **Browser Stations:** We used two 500 MHz Pentium III laptops with 256 MB RAM and running Windows 98 for our browsing stations. Wireless connectivity was provided by 11 Mbps Orinoco PCMCIA WLAN cards. The laptops accessed the streaming formats using Microsoft media, Real and Quicktime players.
- **Monitoring Station:** The packets transmitted from the server to the browser station was passively captured by the monitoring station, which was physically kept close to the browser station and the wireless access point. We used an IBM T21 laptop with 800 MHz Pentium III processor, 256 MB RAM and running Redhat Linux 7.2. Packets were capturing using tcpdump 3.6. We assume that the packets arrive at similar time durations to the tcpdump and the browser applications.

The tcpdump packet traces captured by the Monitoring station were fed to our client-side proxy simulator to an-

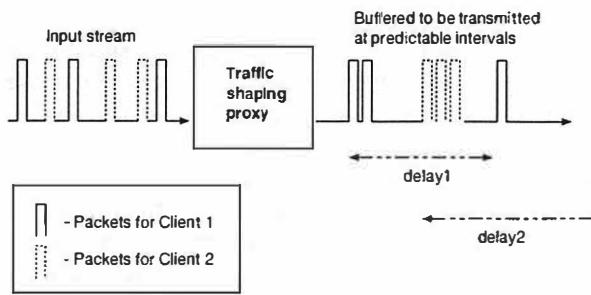


Figure 3: Policies to shape the network traffic to arrive at predictable intervals

analyze the system energy performance without perturbing the Browser stations. We utilized published power parameters [32, 18] for a 2.4 GHz DSSS IEEE 802.11b Orinoco card for our simulations. The model does not simulate lower level energy costs such as unsuccessful attempts to acquire the channel (media contention), or in messages lost due to collision, bit error or loss of wireless connectivity. Further, our simulations do not leverage the battery recovery effect. We assume a linear energy model for wireless NIC power consumption.

### 3.3.1 Multimedia Stream Collection

For our experiments, we used the Wall (movie) theatrical trailer. We replayed the trailer from a DVD player and captured the stream using the Dazzle Hollywood DV Bridge. We used Adobe Premiere 6.0 to convert the captured DV stream to the various streaming formats. The trailer was 1:59 minutes long. The Wall trailer was digitized to a high quality stream and hence allowed us the flexibility of creating streams of varying formats and fidelities. Hence, we use this stream for the rest of this paper.

### 3.4 Traffic shaping policies in the network infrastructure

The various states of packet transmission for a traffic shaping network proxy is illustrated in Figure 3. The proxy buffers network packets from different flows (clients) and transmits them at client-specific intervals. The server maintains separate delay intervals per individual client (e.g. delay1 and delay2) and transmits packets to avoid contention on the wireless network. Such traffic shaping allows multiple clients to operate without interfering with each others' sleep intervals.

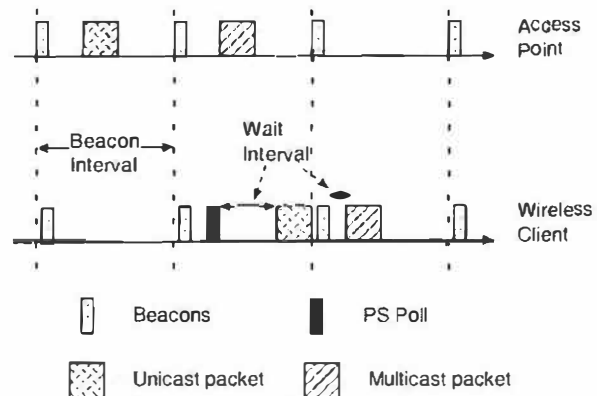


Figure 4: IEEE 802.11 Power Saving Mode (simplified)

## 3.5 Performance metrics

For our experiments, we use the following performance metrics to measure the efficacy of our approach:

- **Energy consumed:** The goal of these experiments is to reduce the energy consumed by the WNIC.
- **Energy metric:** Depending on the delay introduced by our traffic shaping policies, the multimedia players automatically (and incorrectly; since the effective bandwidth available was still the same) adapted to the delays by lowering the stream fidelity. In order to compare the energy consumption in such a scenario (where the amount of data received can be different), we introduce the notion of energy metric; defined as the amount of energy required to download multimedia data (denoted in Joules/KB). It is preferable to decrease the energy metric. In general, even though low fidelity streams consume less overall energy, the energy metric is high as the WNIC's spend most of the time in wasted *idle* or *sleep* states (instead of actually receiving useful data).

## 4 Implications of IEEE 802.11 Power Management in Wireless Access Points

First we explore the effectiveness of IEEE 802.11 MAC level power saving mode for conserving the client WNIC energy consumption. In the next section, we investigate a proxy architecture to effect energy savings.

The IEEE 802.11 wireless LAN access standard [23]

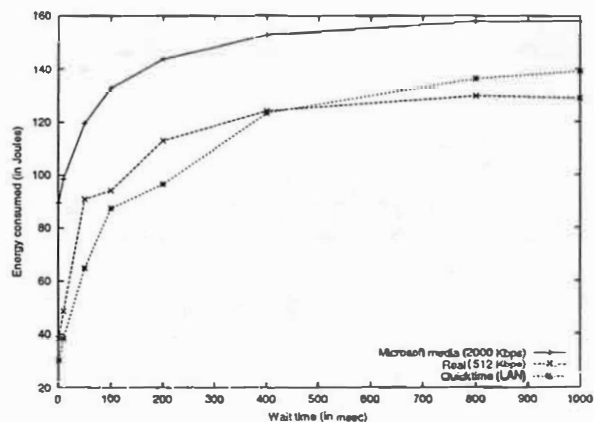


Figure 5: Energy consumed for varying *Wait* times

defines a power saving mode of operation wherein the wireless station and the access point cooperates to conserve energy (illustrated in Figure 4). The wireless station informs the access point of its intention to switch to power saving mode. Once in power saving mode, the wireless station switches the WNIC cards to a lower power *sleep* state; periodically waking up to receive beacons from the access point. The access point buffers any packets for this station and indicates a pending packet using a traffic indication map (TIM). These TIMs are included within beacons that are periodically transmitted from the access point. On receipt of an indication of a waiting packet at the access point, a wireless client sends a PS poll frame to the access point and waits for a response in the active (higher energy consuming) state. The access point responds to the poll by transmitting the pending packet or indication for future transmission. The access point indicates the availability of multiple buffered packets using the *More* data field. For multicast and broadcast packets, the access point transmits an indication for pending packets using a delivery TIM (DTIM) beacon frame; immediately followed by the actual multicast or broadcast packet (without an explicit PS poll from clients). DTIM intervals are usually configurable at the access point and can be any multiple of the beacon interval. The standard does not define the buffer management or aging policies in the access points. Note that the standard does not define a power saving transmit mode for the wireless station itself, it can transmit a packet whenever it wants (regardless of the beacon).

At first glance, it would appear that the 802.11 power saving mode can indeed allow the wireless clients to conserve energy by allowing them to frequently transition to lower power consuming *sleep* state. However, the potential energy saving depends on minimal *Wait* interval (time between the transmission of PS poll mes-

sage and receipt of the data packet; illustrated in Figure 4). The IEEE 802.11 standard does not specify any time bound for this *Wait* interval. For a general purpose wireless access point, reducing the *Wait* interval has the inadvertent side effect of increasing the priority of data packets for wireless stations operating in the power saving mode. Access point manufacturers typically associate power saving mode as a lower throughput state. They also discourage high rate multicast traffic for the same reason.

In order to understand the implications of this wait interval, we developed a simple simulator that modeled the various power saving states of a popular WNIC. We modeled a 2.4 GHz Lucent wireless card that consumes 177 mW, 1319 mW and 1675 mW while in *sleep*, *idle* and *read* states, respectively. These energy parameters were published in [32, 18]. We chose a beacon interval of 100 ms (used by Orinoco access points) and varied the average *Wait* times for reasonable values of 0 through 1000 msec. The access points are modeled with infinite buffer space; as the wait times increase, more packets are buffered and are sent back-to-back in succession. A realistic access point would drop these packets once the buffers fill up.

We plot the results for varying the average *Wait* interval for some of the streams measured in our earlier study ([5]) in Figure 5. For comparison, in [5], we noted that a client-side technique consumes 135 (0.15% data loss), 132 (8% data loss) and 47 (23% data loss) Joules for these MS Media, Apple Quicktime and Real streams, respectively. To receive these streams, the WNICs needed to be in active *read* state for 43.63%, 11.30% and 5.11% of the time, which corresponds to a necessary energy consumption of 90.24, 42.94 and 33.57 Joules, respectively. From Figure 5, we note that the potential energy saving is heavily dependent on the average *Wait* intervals. Wait times of zero illustrates the necessary energy consumption values. For average wait times less than 100 msec, even small increase in average *Waits* can drastically affect the energy saving. However, larger wait times do not offer much energy savings.

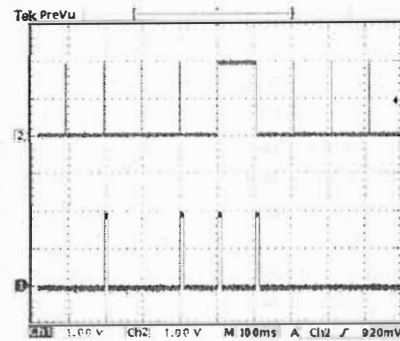
Hence, we tried to measure the actual *Wait* times for typical access points. We carefully disassembled the plastic shielding around a Orinoco Silver PC card and hooked two voltage probes around the two status LEDs. The first LED showed the card power state (transitioning to a high voltage stage while the card is active) while a second LED showed when a data packet was transmitted or received. We connected the PC card to an external Orinoco range extender antenna to reduce the effects of our instrumentation on the proper operation of the wire-



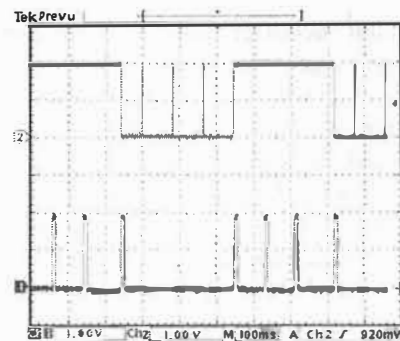
less card. We used this instrumented PC WNIC card on laptops running MS Win 2000, Win 98, Redhat Linux 7.2 and Compaq iPAQ; configured the PC card to operate in power saving mode and watched media streams using MS Media, Real and Apple Quicktime players for the Windows machines, Real for Linux and PocketTV and MS Media players for the iPAQ. For our study, we used Orinoco RG 1000, Orinoco AP 500 and D-Link DWL 1000 access points. All the access points were set up on a dedicated LAN segment (with no background network traffic) and operating on the same wireless channel.

We noticed that Orinoco RG 1000 and AP 500 access points used a beacon interval of 100 msec (in spite of our attempts at changing this interval; the Linux drivers provides an API to request a different beacon interval). The D-Link DWL 1000 allowed us to choose an interval of either 160 msec or 80 msec; the Windows driver did not allow us to modify this interval and chose 80 msec for TIM. We plot several representative results for the card status during our experiment in Figure 6. Note that the different plots show different parts of the video stream. We show the results for several low bitrate streams. These lower quality streams transmit less data and can be expected to offer considerable energy savings. In each graph, the plot for Channel 2 (top) shows the duration that the card stays in active state. Channel 1 (bottom) shows the duration when an actual packet is either transmitted or received at the card. Ideally, we want the top plot to be active for the least amount of time; appearing similar to the bottom plot.

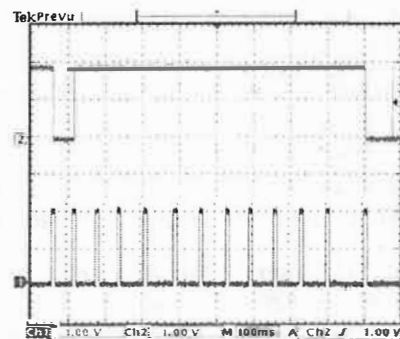
Figure 6(a) plots the results for viewing a MS stream at 56 kbps using the iPAQ device. From Figure 6(a), we note that the power save mode sometimes works optimally. The first two data packets were received with the card in higher power state for the least amount of time. The third packet however triggers the card to stay in higher energy consuming state for a long time, the access point does not transmit the next packet until the next beacon interval (a *Wait* interval of 100 msec). Figure 6(b) plots the results for viewing a Apple Quicktime streaming at 30 kbps from a laptop. For Figure 6(b), we notice similar periods of active waiting. Also in Figure 6(c), we notice longer wait intervals for watching a Real stream at 56 kbps. As noted in [5], Real tends to spend many smaller packets (as compared to Microsoft media). Such packets tend to leave the WNIC at higher energy consuming active state while the access points tries to operate “fairly” for a general audience. In fact, we noticed that watching any stream over 56 kbps tends to completely leave the WNIC in higher energy consuming active state (even though it must be possible to keep the *Wait* times lower, albeit consuming most of the avail-



(a) iPAQ, MS Media stream at 56 kbps, Orinoco AP500 Access point



(b) Windows 98, Apple Quicktime stream at 30 kbps, D-Link DWL 1000 Access point



(c) Windows 2000, Real stream at 56 kbps, Orinoco AP500 Access point

Figure 6: Status of IEEE 802.11b wireless PC card in powersave mode (the top plot (Ch 2) shows the WNIC power state and the bottom plot (Ch 1) shows data transmission/reception intervals. The x-axis shows the time in 100 msec ticks with voltage along the y-axis)

able bandwidth).

We also experimented with the power implications of receiving multimedia streams using multicast packets. The Orinoco drivers allow us to configure the DTIM interval (higher values would reduce the multicast throughput). We observed that even when the DTIM interval was the same as TIM interval, the access points tended to drop the multicast packets. In particular, the D-Link DWL 1000 access points dropped most of the multicast packets. Hence multicast was not a reliable streaming mechanism for our purposes.

In general, we believe that the 802.11b power saving mode has limitations for saving client WNIC energy consumption for the following reasons:

**1. Access point behavior hardware dependent:**

The potential energy saving depends on the policy choices at both the access point and the mobile client station. A general access point needs to balance the need for power saving on a single client with fairly sharing the available bandwidth. Even with no other clients to share the bandwidth, the access points tested still tended to keep the clients waiting for extended periods of time.

**2. Does not co-exist for multiple clients:** For multiple clients accessing multimedia streams simultaneously, the TIM specifies all the clients with pending network packets. The standard does not define the order in which client requests are actually serviced. Clients could wait for the whole time needed to transmit packets for all the clients, even though the PS poll requests were all sent at the same time.

The mobile station can delay the transmission of the initial PS poll message to allow the access point to transmit data packets for other nodes. However, we are not aware of any such protocol defined by the standard to control the client PS poll interval.

**3. Uses fixed TIM interval for all clients:** Even though the standard does not explicitly prevent the access point from dynamically varying the beacon interval to accommodate the observed traffic levels, none of the access points that we tested changed the beacon interval. The choice of the beacon interval is a trade-off between the average packet delay at the access point and the periodicity of client wakeup intervals. As the data rate increases, reducing the TIM interval can reduce the packet delay at the access point. With multiple clients in power saving mode of operation, there is a need for per client TIM intervals.

**4. Power save operation not application aware:**

With the *notification(TIM) → request(PSpoll) → dataatransmission* model of operation, the IEEE 802.11b standard assumes that the data traffic is sporadic and well behaved (few packets spread evenly). On the other hand, multimedia streams tend to be isochronous. Formats such as Real tends to transmit smaller packets at close intervals. Formats such as MS media tends to utilize network fragmentation leading to fragmented packets sent closer to each other. Delaying parts of a fragment can delay the delivery of the entire packet to the multimedia browser.

#### 4.1 Whitecap technology and IEEE 802.11e standard

The upcoming IEEE 802.11e will incorporate the whitecap [9] technology to provide QoS guarantees for multimedia. The technology provides contention free access for multimedia traffic by reserving portions of the transmission spectrum for periodic multimedia traffic. It seems possible for clients to operate in power save mode; transitioning to an active state to receive multimedia streams. The standard is still in the draft stages.

## 5 Implications of energy aware multimedia service

In the last section, we discussed the limitations of utilizing the 802.11b MAC power management scheme for popular streaming media formats. In this section, we explore the implications of modifying the origin server (through a proxy) to customize the streams so that the clients can frequently transition the WNICs to lower energy consuming states.

In general, any traffic shaping at the origin server will be affected by the loss and delay characteristics of the wide-area Internet. Traditionally, buffering at the client had been used to offset these delays. However, the clients need to know the exact time of arrival for the first packet in order to minimize data loss. The loss of the last packet in a stream (which indicates the arrival of the next packet stream) affects the potential energy saving. Also, multiple clients using the same access point but receiving different streams would experience delays because of competing data reception characteristics. In summary, modifying the origin servers to shape the network traffic

Table 1: Energy consumed and % packets dropped by a client-side history based approach (detailed discussion in [5])

Stream Format	Stream b/w (in Kbps)	Energy (in Joules)	Client-side adaptation	
			Energy (in Joules)	Bytes dropped (%)
Microsoft Media	56	157.6	35	1
	128	157.7	52	2
	256	160.2	60	0.5
	768	163.2	70	0.25
	2000	174.8	135	0.15
Real	56	119.2	116	4
	128	119.2	82	11
	256	124.3	120	5
	512	133.2	132	8
Quicktime	56	149.7	41	30
	128	150.1	38	38
	256	149.2	47	23

can enable clients to frequently transition to lower power states has the following drawbacks:

- Loss of the control packet in a stream can adversely affect the energy savings. Throughout this work, we assumed that the wireless network does not suffer from noticeable multimedia data packet loss. In general, if the control packet specifying the client sleep interval from the local proxy is lost, then the client will wait in a higher power consuming *idle* state. Losing the data packets itself would trigger high level mechanisms that adapt the stream to a lower fidelity stream.
- Multiple local clients receiving streams from different servers can lead to conflicting schedules on the network.
- Any packet delay in the wide area can leave a client in a higher power consuming state.

We implemented our policies on a local proxy based architecture. The local proxy buffers the multimedia packets and periodically transmits them to the client (similar in spirit to the IEEE 802.11 beacons). The local proxy could also dynamically inform the client-side of the next packet arrival time using a special control packet. The client-side proxy uses the interval between transmissions to transition the client WNIC to a lower energy consuming *sleep* state. The client-side proxy informs the local proxy server of the access point that it is associated with. The local proxy schedules the transmissions in such a way to avoid conflicts with other clients using the same wireless access point.

First we analyze the energy savings for the various popular streaming formats (MS Media, Real and Apple

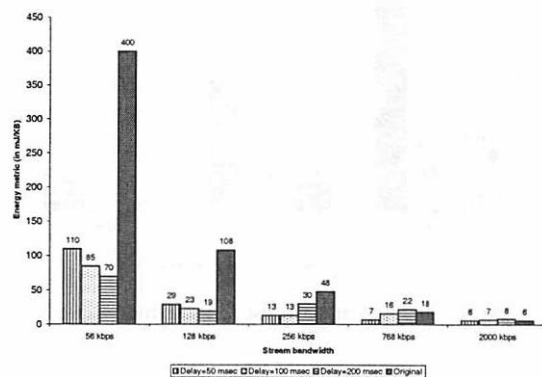


Figure 7: Energy metric for MS media streams

Quicktime) using streams customized for different network bandwidth requirements. We also explore the implications of multiple clients sharing the same wireless access point. For reference, the energy savings and the associated data loss for the various streaming formats using a client based adaptation strategy (that was discussed in [5]) is tabulated in Table 1.

## 5.1 Single wireless client associated with the wireless access point

First we perform experiments to explore the energy saving possible with a cooperating proxy that enables the clients to transition to lower energy consuming *sleep* state. We explore the implications for popular streaming formats (Microsoft media, Real, Apple Quicktime). We configured the local proxy to transmit buffered packets after a delay of 50, 100 and 200 msec.

In some cases (e.g. high fidelity streams), depending on

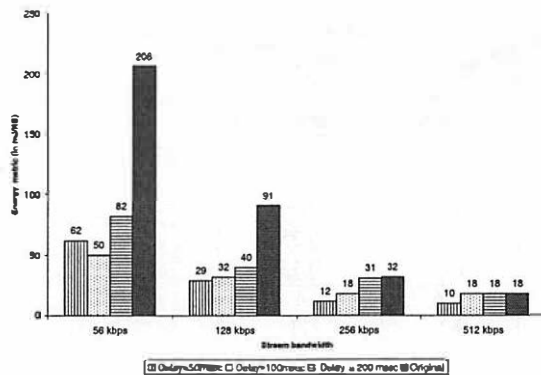


Figure 8: Energy metric for Real streams

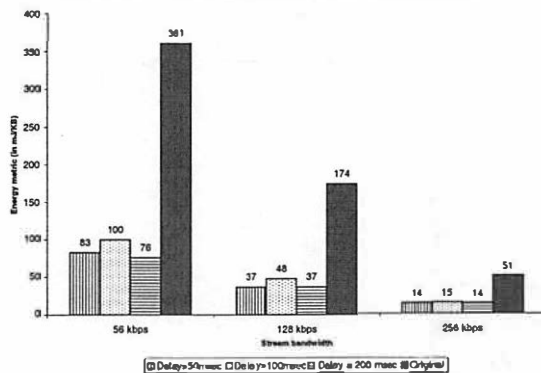


Figure 9: Energy metric for Quicktime streams

the delay introduced, the multimedia players automatically (and incorrectly) adapted to the delays by lowering the stream fidelity. In order to compare the energy consumption in such a scenario, we used the notion of energy metric, defined as the amount of energy required to download multimedia data (denoted in Joules/KB). It is preferable to reduce the energy metric.

### 5.1.1 Microsoft media streaming format

We plot the energy metric for video streams customized for 56 kbps, 128 kbps, 256 kbps, 768 kbps and 2000 kbps bandwidth streams in Figure 7. From Figure 7, we note that a proxy that transmits packets every 200 msec can offer energy savings for low fidelity stream streams (without any data loss associated with a client-only scheme [5]). For a 56 kbps stream, a server that transmits packets every 200 msec can reduce the energy consumption by as much as 83%. However, for high bandwidth streams (low bandwidth streams are themselves transmitted infrequently) and increased delays, the media player adapts to increasing delays by reducing the stream fidelity. This results in reduced energy consumption while increasing the energy metric. In fact,

the energy metric can sometimes be worse than (e.g. 768 kbps stream) receiving the original unmodified streams. Such adaptation can be counteracted by buffering the network packets in the client-side proxy and locally delivering them to the multimedia player at a more regular pace. Note that such additional buffering introduce their own energy requirements for maintaining the local buffers.

Also, we used the *nanosleep()* system call in the local proxy to delay packets for delivery. General purpose operating systems such as FreeBSD and Linux schedule sleeping jobs at 10 msec intervals and hence the actual sleep intervals can be at least 10 msec more than the programmed value. We noticed that this extra interval tends to be more than 10 msec for processes that sleep for longer intervals of time. The client-side proxy actively waits for packets in this extra 10 msec in a higher energy consuming *idle* state, leading to increased energy consumption and higher energy metric. Real time schedulers for Linux can reduce this scheduling interval to 2 msec. We are currently investigating such schedulers to further reduce the energy consumption.

### 5.1.2 Real streaming format

We repeat the experiments from last section for Real streams transmitted at 56 kbps, 128 kbps, 256 kbps and 512 kbps and plot the corresponding energy metric in Figure 8. We configured the real player to not transmit the stream reception quality feedback to the origin servers and hence the system did not adapt to the packet reception delays. We noted that the server assisted policies offer substantially better energy saving than simple client-side only policies without any associated data loss. For example, a 56 kbps stream with a local proxy that transmits packets every 100 msec only consumed 28 Joules (as compared to 116 Joules and 4% data loss for client-side history based mechanisms). Also, recall that Real typically streams the video streams quicker than the other formats. Introducing high delays (e.g. 200 msec) prolonged the stream transmission, adding extra *sleep* cycles and slightly increasing the energy metric.

### 5.1.3 Quicktime streaming format

In the last two sections, we explored the potential energy saving in a server that transmits the stream packets in predictable intervals for Microsoft media and Real streams. In this section, we repeat the experiments for

Table 2: Energy consumed and % packets dropped by the client-side history based approach for 2 simultaneous clients in the same wireless access point

Stream Format	Stream b/w	Energy (in Joules)	Bytes dropped (%)
Microsoft Media	56 Kbps	37.00	0.79
	128 Kbps	54.03	0.09
	256 Kbps	57.98	51.77
	768 Kbps	72.74	19.77
	2000 Kbps	135.34	10.17
Real	56 Kbps	39.63	39.10
	128 Kbps	58.23	24.74
	256 Kbps	79.29	21.49
	512 Kbps	96.35	34.42
Quicktime	56 Kbps	27.17	29.71
	128 Kbps	27.96	49.22
	256 Kbps	32.94	41.49

Apple Quicktime streams at 56 kbps, 128 kbps and 256 kbps and plot the results in Figure 9. From Figure 9, we notice the potential energy savings without the associated high data loss rates (illustrated in Table 1). Note that Quicktime transmits the audio and video portions of the stream using separate UDP stream channels. The local proxy needs to be aware of these independent streams in order to schedule the data transmissions.

## 5.2 Implications of multiple clients using the same wireless access point

In the last section we explored the potential energy savings for a simple server policy that transmits packets at predictable intervals (similar to the IEEE 802.11 MAC level power saving mode). We performed our experiments on a dedicated WLAN environment with a single wireless client. We showed that the potential energy savings over client-side policies. We discussed the effects of operating system scheduling policies that can reduce the potential energy saving. We also identified media player adaptation to increased packet delay and their effects on energy metric.

However, in a typical operating scenario, there could be many mobile clients within a single access point sharing the same physical medium. The local proxy with the knowledge of the physical WLAN limitation can schedule the clients such as to minimize data contention at the network. In this section, we explore the implications of two clients using the same wireless access point, consuming the same video stream (slightly offset in time to avoid the same data from being multicast just once).

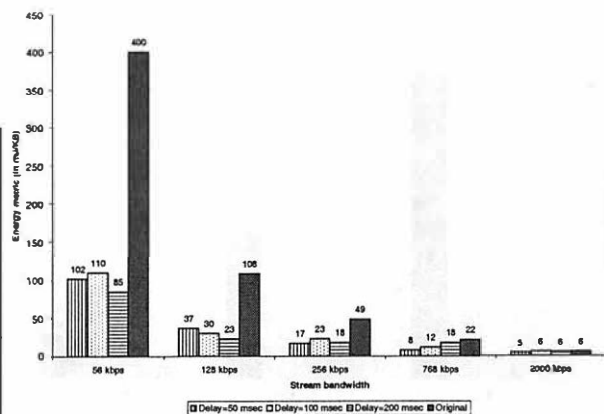


Figure 10: Energy metric for two simultaneous MS media streams

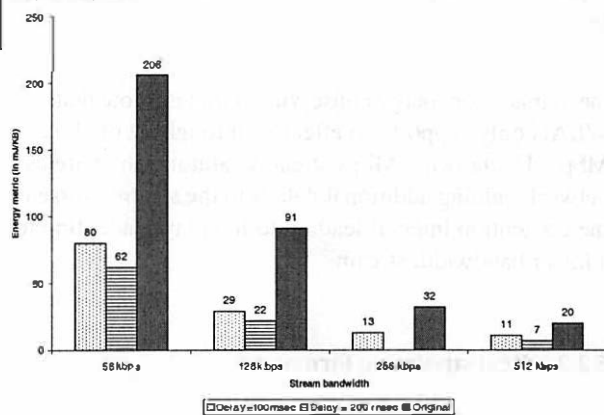


Figure 11: Energy metric for two simultaneous Real streams

First we tabulate the energy saving and the associated data loss for a client-side history based policy (described in [5]) in Table 2. Table 2 shows the inherent limitations of a client-side history based policies in a network with high contention. Both Real and Quicktime streams as well as high bandwidth Microsoft media streams experience higher data loss compared to a single client case shown in Table 1. Low bandwidth Microsoft media streams are transmitted at fairly regular intervals which are not affected much by the increased network contention.

### 5.2.1 Microsoft media streaming format

We performed experiments with a local proxy servicing two clients accessing the same stream of identical stream quality. We plot the energy metric for the various Microsoft media streams in Figure 10. We notice similar results as in the single client case (Figure 7), showing the potential for an application level contention reduction

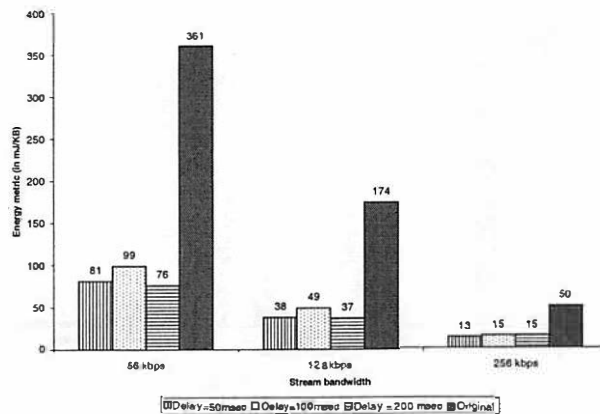


Figure 12: Energy metric for two simultaneous Quicktime streams

mechanism for energy conserving policies. Note that our WLAN only supports an effective throughput of about 4 Mbps. Using two 2 Mbps streams saturates the wireless network; adding additional delays to the streams worsens the contention interval leading to the player adapting to a lower bandwidth stream.

### 5.2.2 Real streaming format

We repeated the experiments for Real streams and plot the results in Figure 11. For the most part, two clients provide similar savings as a single client case. However, Real players inexplicably experience fatal errors particularly operating with a delay of 50 msec. We are presently investigating why the Real players crash when two players are simultaneously accessing two multimedia streams.

### 5.2.3 Quicktime streaming format

We continue with our analysis for the various Quicktime streams and plot the results in Figure 12. From Figure 12, we notice similar performance gains for two clients simultaneously accessing multimedia streams. Again, we notice that Quicktime clients adapt to any introduced delays by lowering the stream quality. A caching client-side proxy can offset this client behavior.

In this section, we showed the effectiveness of application aware local proxy in shaping the multimedia traffic. The system can not only allow the clients to better manage their energy, but also schedule the packets to avoid local WLAN network contention. The Quicktime multimedia players adapt to any introduced delays by lower-

ing the stream quality. A caching client-side proxy can help offset such client behavior.

## 6 Discussion

In this paper, we show the limitations of IEEE 802.11 power saving mode for receiving popular multimedia streams (Microsoft media, Real and Quicktime). We showed that the non-deterministic TIM interval and the associated *Wait* interval can adversely affect the potential energy savings. We showed that the WNICs effectively switch out of the power saving modes for even moderately high bandwidth streams (128 kbps). Also, a single TIM can reduce the energy savings for multiple clients contending for the same TIM beacons by increasing the wait intervals for each stream.

On the other hand, an application-specific server side traffic shaping mechanism can offer good energy saving for all the stream formats without any data loss. We use a simple server enhancement to transmit network packets at predictable intervals. We show that we can reduce the energy metric (Joules/KB) by as much as 83%. We show how our approach can offer similar benefits for two clients sharing the same wireless access point. We note that the operating system scheduling mechanisms induce additional latencies that reduce further potential energy savings. Also, Quicktime players adapt to any network delays by lowering the stream quality.

Our work makes the following contributions towards the design of such application specific energy-aware network traffic shaping mechanisms:

- We show that the amount of energy saving delays introduced depends on the stream requirements; lower fidelity streams are more tolerant to longer delays.
- Operating system scheduling mechanisms can restrict choosing too small values of these delays.
- These mechanisms have to take the stream format into consideration. Formats such as Quicktime that transmit a given media stream using at least two independent channels (one each for audio and video) should be treated properly so as to avoid conflicts within the same application.
- Additional information such as the associated access point can also help avoid network media contention issues.



- Future media players should provide a configurable mechanism for specifying wireless networks such that these energy saving transitions do not trigger network congestion response.

## Acknowledgments

We thank Ben Bishop for his help with the oscilloscope. We especially thank our reviewers and our shepherd Mary Baker for all their helpful comments and suggestions. This work was supported in part by a research grant from the Yamacraw initiative.

## References

- [1] Prathima Agrawal, Jyh-Cheng Chen, Shaline Kishore, Parameswaran Ramanathan, and Krishna Sivalingam. Battery power sensitive video processing in wireless networks. In *Proceedings IEEE PIMRC98*, Boston, September 1998.
- [2] Apple Quicktime. <http://www.apple.com/quicktime/>.
- [3] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile Memory for Fast, Reliable File Systems. In *Proceedings of the 5th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, October 1992.
- [4] The Bluetooth SIG, <http://www.bluetooth.com/Specification of the Bluetooth system, v 1.1 edition>, February 2001.
- [5] Surendar Chandra. Wireless network interface energy consumption implications of popular streaming formats. In Martin Kienle and Prashant Shenoy, editors, *Multimedia Computing and Networking (MMCN'02)*, volume 4673, pages 85–99, San Jose, CA, January 2002. SPIE - The International Society of Optical Engineering.
- [6] Surendar Chandra, Carla Schlatter Ellis, and Amin Vahdat. Managing the storage and battery resources in an image capture device (digital camera) using dynamic transcoding. In *Proceedings of the Third ACM International Workshop on Wireless and Mobile Multimedia (WoWMoM'00)*, pages 73–82, Boston, August 2000. ACM SIGMOBILE.
- [7] Carla Fabiana Chiasserini and Ramesh R. Rao. Pulsed battery discharge in communication devices. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking (MOBICOM '99)*, pages 88–95, Seattle, WA, August 1999.
- [8] Sukjar Cho. Power Management of iPAQ, February 2001.
- [9] Cirrus Logic. Whitecap2 wireless network protocol - white paper. Technical report, Cirrus Logic, 2001.
- [10] Mark D. Corner, Brian D. Noble, and Kimberly M. Wasserman. Fugue: time scales of adaptation in mobile video. In *Proceedings of the SPIE Multimedia Computing and Networking Conf.*, San Jose, CA, January 2001.
- [11] Anindya Datta, Aslihan Celik, Jeong Kim, Debra E. VanderMeer, and Vijay Kumar. Adaptive broadcast protocols to support power conservant retrieval by mobile users. In *Proceedings of Data Engineering Conf. (ICDE)*, pages 124–133, September 1997.
- [12] Fred Douglass, P. Krishnan, and Brian Bershad. Adaptive Disk Spin-down Policies for Mobile Computers. In *2nd USENIX Symposium on Mobile and Location Independent Computing*, April 1995. Monterey CA.
- [13] Carla S. Ellis. The case for higher-level power management. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, March 1999.
- [14] Keith I. Farkas, Jason Flinn, Godmar Back, Dirk Grunwald, and Jennifer M. Anderson. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. In *ACM SIGMETRICS*, pages 252–263, Santa Clara, CA, June 2000.
- [15] Laura Marie Feeney and Martin Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *Proceedings IEEE INFOCOM 2001*, volume 3, pages 1548–1557, Anchorage, Alaska, April 2001.
- [16] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles (SOSP)*, pages 48–63, December 1999.

- [17] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU. In *Proceedings of 1st ACM International Conference on Mobile Computing and Networking (MOBICOM95)*, pages 13–25, November 1995.
- [18] Paul J. M. Havinga. *Mobile Multimedia Systems*. PhD thesis, Univ. of Twente, February 2000.
- [19] David P. Helmbold, Darrell E. Long, and Bruce Sherrod. A Dynamic Disk Spin-Down Technique for Mobile Computing. In *Proceedings of the 2nd ACM International Conf. on Mobile Computing (MOBICOM96)*, pages 130–142, November 1996.
- [20] Tomasz Imielinski, Monish Gupta, and Sarma Peyyeti. Energy Efficient Data Filtering and Communications in Mobile Wireless Computing. In *Proceedings of the Usenix Symposium on Location Dependent Computing*, April 1995.
- [21] Robin Kravets and P. Krishnan. Power Management Techniques for Mobile Communication. In *Proceedings of the 4th International Conf. on Mobile Computing and Networking (MOBICOM98)*, pages 157–168, October 1998.
- [22] Robin Kravets, Karsten Schwan, and Ken Calvert. Power-aware communication for mobile computers. In *International Workshop on Mobile Multimedia Communications (MoMuc'99)*, November 1999.
- [23] LAN/MAN Standards Committee of the IEEE Computer Society. *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE, 3 Park Avenue, New York, NY 10016, 1999.
- [24] LAN/MAN Standards Committee of the IEEE Computer Society. *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Higher-speed Physical Layer Extension in the 2.4 GHz Band*. IEEE, 3 Park Avenue, New York, NY 10016, 1999.
- [25] LAN/MAN Standards Committee of the IEEE Computer Society. *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Higher-speed Physical Layer Extension in the 5 GHz Band*. IEEE, 3 Park Avenue, New York, NY 10016, 1999.
- [26] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *USENIX Association Winter Technical Conf. Proceedings*, pages 279–291, 1994.
- [27] Jacob Lorch and Alan J. Smith. Software Strategies for Portable Computer Energy Management. *IEEE Personal Communications Magazine*, 5(3):60–73, June 1998.
- [28] Microsoft Windows Media Technologies. <http://www.microsoft.com/windowsmedia/>.
- [29] Real Player. <http://www.real.com/>.
- [30] Suresh Singh, Mike Woo, and C. S. Raghavendra. Power-Aware Routing in Mobile Ad Hoc Networks. In *The Fourth Annual ACM/IEEE International Conf. on Mobile Computing and Networking*, pages 181–190, 1998.
- [31] Cliff Skolnick. 802.11b community network list. <http://www.toaster.net/wireless/community.html>, <http://www.toaster.net/wireless/aplist.php>, 2001.
- [32] Mark Stemm, Paul Gauthier, Daishi Harada, and Randy H. Katz. Reducing power consumption of network interfaces in hand-held devices. In *Proceedings of the 3rd International Workshop on Mobile Multimedia Communications (MoMuc-3)*, Princeton, NJ, September 1996.
- [33] Amin Vahdat, Alvy Lebeck, and Carla Schlatter Ellis. Every joule is precious: The case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.
- [34] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for Reduced CPU Energy. In *USENIX Association, Proceedings of First Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994. Monterey CA.
- [35] John Wilkes. Predictive Power Conservation. Technical Report HPL-CSP-92-5, Hewlett-Packard Labs, February 1992.

# Characterizing Alert and Browse Services for Mobile Clients

Atul Adya, Paramvir Bahl, Lili Qiu  
Microsoft Research

1 Microsoft Way, Redmond, Washington 98052  
{*adya, bahl, liliq*}@microsoft.com

## Abstract

There is a fair amount of evidence that suggests that Internet access from wirelessly-connected mobile handheld devices is gaining popularity. However, there haven't been too many studies that have focused solely on analyzing the wireless Internet. In this paper, we study the notification and browse services provided by a large commercial web site designed specifically for users who access it via their cell-phones and PDAs. Unlike previous web studies that have analyzed browse services provided over wired networks, we focus primarily on browse and notification services provided over wireless channels. Specifically, we analyze the notification and browser traces to understand the system load, the type of content accessed, and user behavior. We discuss the implications of our findings for techniques such as multicast, query caching and optimization, and transport protocol design.

## 1 Introduction

Over the last decade the cellular phone industry and the World Wide Web have experienced a phenomenal growth as people around the world have embraced these technologies at a remarkable rate. Today, most major wireless service providers in the United States, Europe, and Japan offer wireless Internet services and many Internet companies provide content that has been adapted to suit the limited display, bandwidth, memory, and processing power of small devices.

Another emerging trend, related to wireless Internet, has to do with how users manage the gigantic information flow that the Internet provides. Realizing that users are being overwhelmed with information, several web content providers allow users to switch their data access model from polling and navigation to notifications or alerts. Instead of periodically browsing through the web sites for potentially useful information, an increasing number of users are adopting the model where they reg-

ister for information in which they are interested. These users provide a callback address usually in the form of an email address, a cell-phone number, or a pager number, depending on their perceived importance of the information. Whenever the relevant event is triggered, the content provider sends a notification to the user. Examples of some US companies that provide such notifications include Yahoo Mobile, MSN Mobile, AOL Anywhere, and InfoSpace. All of these services allow users to subscribe to alerts for stock quotes, sports scores, lottery, horoscope, calendar events etc. If alert services becomes a popular form of user interaction with the web, it will be critical for content provider and content management companies to handle these notifications efficiently. Proper management of notifications involves understanding which types of notifications are popular, which types of devices are used by subscribers for receiving notifications, the frequency of sending these notifications on a per user basis, etc.

In this paper, we study notification and browse services provided by a large popular commercial web site that is designed specifically for US users who access it via their cell-phones and PDAs. Unlike most previous web studies, which have analyzed browsing services provided over wired networks, we focus primarily on a web server that delivers notification and browsing services over wireless channels. We analyze notification and browser traces to understand the system load, the type of content that is accessed, and user behavior. We believe that our study is important for content providers, wireless ISPs, and web site managers.

We note here that we do not study the performance of the web server subsystem or its architectural design. Instead, we use web server logs to analyze the browse and notification patterns of wireless web users.

The rest of this paper is organized as follows. In Section 2 we review previous work done in the field of web trace analysis. In Section 3, we describe the different ways in which the web site is accessed, the characteris-

tics of the data logs, and the types of analyses we carry out. We present detailed analysis of the notification and browse logs in Sections 4 and Section 5, respectively. In Section 6, we examine the degree of correlation between the usage of browse and notification services. We conclude in Section 7.

## 2 Related Work

There have been a number of studies on the access dynamics of web servers servicing clients over a wired network. These studies include analyses of web access traces from the perspective of proxies [7, 20, 21], browsers [6, 9], and servers [4, 16]. However, to our knowledge, all previous web workload studies have been conducted for browse services only and there are no published studies on notification services. Consequently, we believe, our analysis of notification services is the first study of its kind.

Even for the browsing services, most studies analyze web servers serving clients over wired networks. There are very limited studies on web servers serving clients over wireless channels. The study closest to ours is the one done by Kunz *et al.* [12], which analyzes network traces generated by a mobile browser application. Specifically, their paper analyzes user behavior (bytes transferred and time spent on the wireless link) based on the notion of a session that was chosen to be 90 seconds; however, a different session period could potentially change their results. The main limitation of their work is the size of the data analyzed: although the traces were collected over a period of seven months, only 80K entries were logged. It is unclear whether the inferences drawn from this study can scale up to large commercial sites. In contrast, we analyzed traces with millions of entries generated over a period of 12 days at a large commercial site. Furthermore, their study also has the limitation that it uses client IP addresses for identifying users; since IP addresses can be reassigned to different users, it is difficult to perform an accurate user-based analysis. In our study, since every entry in the logs contains a unique identifier for every access/notification, we are able to carry out user-behavior analysis more accurately. In addition, our study is broader as we focus on user behavior, server load, content, and document popularity analysis.

Tang and Baker analyzed a seven-week trace of a metropolitan-area packet radio wireless network, and a twelve-week trace of a building-wide local-area wireless network [18, 19]. Both studies focus on how the networks were used, e.g., when the networks were most ac-

tive, how active the network were, and how often users moved, etc. They did not consider the content or applications for which people used the wireless networks, which is the focus of our paper.

Recently, Balachandran *et al.* [5] analyzed the user behavior and network performance of an IEEE 802.11 based wireless local area network (LAN) using a workload captured at a three day technical conference event. Their study focused on characterizing wireless LAN users for the purpose of coming up with a parameterized model to describe them. Additionally, they carried out workload analysis to address the network capacity planning problem. Their study is very different from ours in terms of analysis, methodology and objectives. While we focus primarily on wireless browse and notification services, they consider all network traffic for improving the network performance. Furthermore, the data-set they captured and analyzed is smaller and significantly different from the web server traces we analyze.

In the sections that follow, whenever appropriate, we refer to related work done by other researchers and compare it with our findings.

## 3 Data Characteristics

Before presenting the analysis, we briefly describe the different ways in which the web site is accessed, the characteristics of the data logs, and the types of analyses we carried out.

For the web server we used in this study, a single browse request results in exactly one HTTP request to the server. There are no images or other types of content embedded in the page that is transmitted to the client as a result of this request.

In the rest of the paper, we use the term *notification document* to refer to a unique document that may be sent to multiple users; we refer to each such transmission as a *notification message*, which includes duplicates.

### 3.1 Types of Accesses

For browsing, the web site is accessed in three different ways and we categorize the browse accesses based on this usage: *desktop*, *offline*, and *wireless*. Desktop accesses include requests from desktop and laptop machines connected to the web site via wireline networks. Offline accesses are generated due to handheld devices such as PDAs. Companies such as Avantgo and Vindigo

offer services that let users select content from different web sites and download it onto a handheld device for browsing at a later time. The content download occurs when a user synchronizes his/her handheld with a desktop machine and is controlled by a “downloader” program; we refer to these programmatic accesses by the downloader as offline accesses. Wireless accesses occur due to browse actions initiated by users from their cell-phones or wireless devices. Typically, a request from a cell-phone is directed to a “gateway” (operated by the user’s service provider) that forwards the message to the web site; this gateway also forwards the reply back to the cell-phone. Thus, from the web site’s perspective, it just communicates directly with the gateway machines using the standard HTTP protocol. Since one gateway can serve multiple clients, we do not use IP addresses to identify users; instead, we use a unique identifier assigned to every client that is logged with each access.

Browser Type	No. of accesses	No. of users
Desktop	7,342,206	639,971
Wireless	2,210,758	58,432
Offline	20,508,272	50,968
Misc	2,944,708	1,634

Table 1: User accesses according to browser types

We determine the type of access based on the browser type stored in the log entry corresponding to that access. For example, entries with browser type “Mozilla Windows”, “Avantgo”, “UP.Browser” are categorized as desktop, offline and wireless accesses respectively. In Table 1 we show the number of accesses according to the browser type (in our case, each access corresponds to a single HTML page). The last category (*Misc*) corresponds to log entries for which the browser type either was empty or contained characters that could not be mapped to any known browser client. The table also shows the number of unique users that were responsible for different types of accesses. Note, the number of desktop users is much higher than the offline and wireless users due to the fact that a large number of users use their desktop machines to register with the web site.

In the case of notifications, there is a client type in the logs that tells us the type of the registered clients. More than 99% of the messages were sent to wireless clients; the remaining were sent to desktop clients.

### 3.2 Description of Data Logs

We had access to logs for 12 days of web browsing from August 15, 2000 through August 26, 2000. There were

approximately 33 million entries in the browse logs. Additionally, we used notification logs from August 20, 2000 through August 26, 2000, which contained 3.25 million entries. For our analysis of the correlation between browse and notification services (Section 6), we obtained additional notification logs and performed the comparison for the period from August 15, 2000 through August 26, 2000.

When a registered user sends a browse request to the web server, a unique identifier corresponding to the user is sent to the server and logged in the web traces (for unregistered users, the id field is empty). We use these identifiers for performing the user-based analysis. Each log record also contains other pieces of useful information along with the user ids, such as the date, time, type of browser, the URL accessed, the data received and sent by the server, etc.

When a notification message is sent, a record is logged in a database. We obtained a part of this database for our analysis. The database entries contained information about the server from where the notification message was sent, a user id, type of the device to which the message was sent (e.g., phone or pager), type of alert, when it was sent, etc.

To efficiently manipulate a large amount of data logs (over 10 GB), we consolidated them into a commercial database system and created indices on columns such as date, user id, and URL. To overcome the limited expressiveness of our database language (in terms of string manipulation), we further processed the database output using Perl scripts.

### 3.3 Types of Analyses

We now discuss the types of analyses that we perform on the notification and browse logs, and the motivations for doing these analysis.

1. **Content analysis:** We are interested in questions such as: (i) what are the most popular content categories, and (ii) what is the distribution of message sizes? We believe such questions are important to (i) content providers who need to understand better how to prioritize and use the system and network resources efficiently, and to (ii) web site developers who are interested in supporting fast access to popular content.
2. **Popularity analysis:** We are interested in the popularity distribution of notification and browse doc-

uments. In particular, we are interested in comparing these accesses to the well-known Zipf-like distribution as reported in previous web studies [4, 7, 10, 14, 16], and in determining how concentrated are the number of requests/transmissions for popular documents. This has significant implication for the effectiveness of web caching and multicast delivery.

3. **User-behavior analysis:** We are interested in classifying users according to their access patterns. This is useful for personalization, targeted advertising, prioritizing, and capacity planning. Specifically, we look at the following aspects of user behavior:

- *Spatial Locality:* whether users in the same geographical region tend to receive/request similar notification and browsing content.
- *Temporal Stability:* whether users are interested in browsing similar documents over time.
- *User Load Distribution:* how different users place load on the web site; for service providers, this distribution has implications on pricing.

## 4 Notification Log Analysis

Table 2 shows the overall statistics for the notification logs. In one week, the server sent out 3.25 million notification messages for a total of 295 megabytes. One fourth of the messages sent out were distinct, while the remaining messages had the same content but sent to different users (in some cases, the same message is sent to a user multiple times, e.g., if a user has registered for information to be delivered at specific times and the information has not changed during that period). The significant amount of duplication in messages sent to different users suggests that sending notification via application-level multicast would be useful; Section 4.2 examines this issue in greater depth. There were 200,860 distinct users, of which 99.02% were wireless users. The notifications were sent at the average rate of 323 messages per minute. The peak rate was much higher, approximately 30 times as high as the average rate.

### 4.1 Content Analysis

We begin our analysis by looking at the content of the notifications sent to various users.

Total messages	3,251,537
Total distinct messages	884,272
Total bytes transmitted	295 MB
Total bytes of unique messages transmitted	71.3 MB
Total number of users	200,860
Total number of wireless users	198,882
Avg. notification rate	322.57 (msgs/min)
Peak notification rate	9502 (msgs/min)

Table 2: Overall statistics for the notification logs for the period from Aug 20 through Aug 26, 2000.

#### 4.1.1 Popular Categories

We classified the notifications into categories based on the subject field, which was recorded in the notification logs. We plotted the number of messages sent for each notification category in Figure 1, and the number of users who received the notification message for each category in Figure 2.

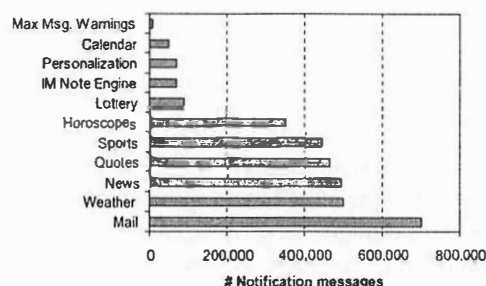


Figure 1: The total number of notifications sent for each category.

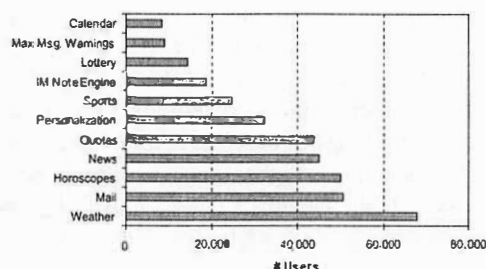


Figure 2: The total number of users who received notifications for each category.

As Figure 1 shows, email, weather, news, stock quotes, sports, and horoscopes are the most popular categories in terms of the total number of notification messages. In comparison, weather, email, horoscopes, news, and stock quotes are the most popular categories in terms of the total number of users (see Figure 2). As we had expected, email alerts were very popular. On the other hand, we had not expected weather-related notifications



to be so popular. Intuitively, one might have expected stock quotes and news to be more popular, especially since users have to explicitly register for different notification types (including weather), i.e., notifications are not being sent due to some default setting on the user-signup page. Another surprise was the low popularity of calendar alerts. For calendar alerts, it is possible that subscribers use handheld devices that are not connected to the wireless Internet, for example, PDAs with pre-installed software to handle scheduled meetings, anniversaries, etc.

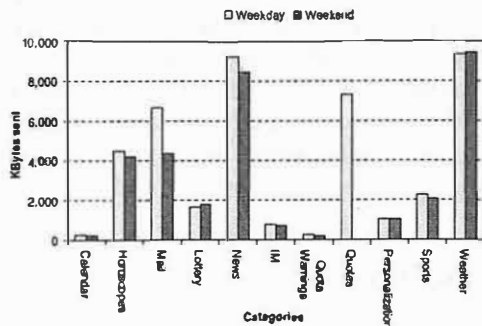


Figure 3: Change of user interest between weekday and weekends

Next we analyzed how user interest changed during the course of a week. Figure 3 shows a comparison between the amount of notification data sent on a weekday versus a day on the weekend. As one would expect, there is a significant difference between the number of stock quote alerts sent during the weekday compared to those sent on the weekend. Similarly, there are fewer mail alerts on weekends; this is probably due to lower levels of work activity that occur on weekends relative to weekdays, resulting in fewer triggering events. For other categories (e.g., sports, weather, horoscopes), the number of notification messages does not vary significantly over weekends and weekdays. We attribute these patterns to the fact that not many users personalize all aspects of their notification portfolio in a very fine-grained manner (for event types such as weather, the web site allows users to select the frequency and the time of delivery).

#### 4.1.2 Notification Message Size and Its Implications

We find that notification messages are small. Specifically, all messages contain less than 256 bytes. We show the message size distribution in Figure 4 to illustrate this point. Consequently, it is important for the delivery protocol to handle small messages efficiently. For example, if the protocol creates a new TCP connection for every notification message, the overhead can be high. In par-

ticular, the connection establishment may increase the user-perceived latency by a factor of 3 (i.e., from one half round-trip time to one and a half round-trip time). Assuming the average notification message size to be 128 bytes, the connection setup and tear-down increases the bandwidth usage from 168 bytes per message to 448 bytes per message (i.e., 7 additional packets: 3 packets in the three-way handshake connection setup, and 4 packets in the connection teardown).

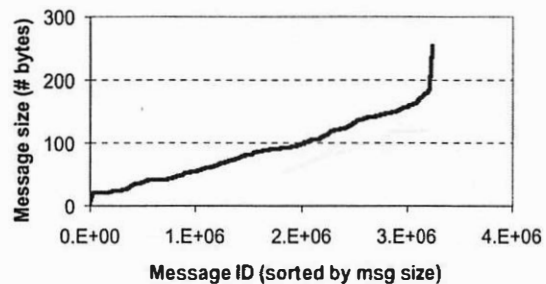


Figure 4: Size distribution of notification messages (including duplicates).

One suggestion for reducing the overhead of connection setup and teardown is to use persistent connections [13], i.e., reuse a TCP connection for multiple transfers. In our case, the servers sending the notification messages can maintain persistent connections with the gateways of the wireless ISPs and then send all messages on this connection.

#### 4.2 Message Popularity Analysis and Its Implications

Several studies have found that web accesses follow Zipf-like distribution: the number of requests to the  $i^{th}$  most popular object is proportional to  $\frac{1}{i^\alpha}$  [3, 4, 6, 7, 10, 14, 16]. The estimates of  $\alpha$  range from 0.5 to 1 for web proxy logs [7, 10, 14], and range from 1 to 2 for web server logs [4, 16]. It is interesting to examine whether notification messages exhibit a similar property.

To do the above, we take the following approach: For each notification document, we count the number of notification messages (i.e., copies) that were sent on a given day. We plot the total number of transmissions of a document (i.e., notification messages) versus the popularity ranking of the document on a log-log scale. Figure 5 shows the plot for August 21, 2000. The plots for the other days are similar, and are omitted for brevity. If we ignore the first few notification documents and the flat tail in Figure 5 (as is done in the previous work [6, 7, 16]), we note that the curve fits a straight line reasonably well. We compute the values of  $\alpha$  using least-square fitting, after excluding the top 20 doc-

uments and the flat tail (the latter set represents the notification documents that were sent only once or twice). The straight line on the log-log scale implies that the notification documents follow a Zipf-like distribution. We find that for our complete data-set the value of  $\alpha$  varies from 1.137 to 1.267 (in Figure 5, the value of  $\alpha$  is 1.146). These values are higher than the  $\alpha$  in the web proxy logs [7, 10, 14], and lower than (but close to) the  $\alpha$  observed for popular web server logs [16].

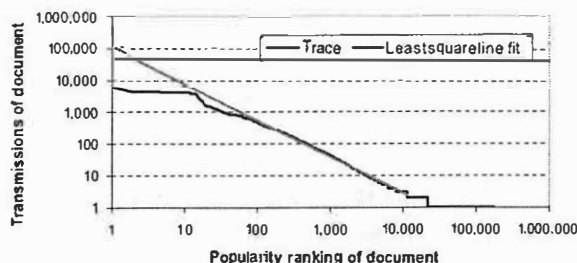


Figure 5: Frequency of notification documents versus ranking in log-log scale (for August 21, 2000).

Figure 6 shows the cumulative distribution of notification documents on August 21, 2000. The top 1% of notification documents (i.e., 1704) account for 54.24% of the total notification messages. In the logs for other days, the top 1% of notification documents account for 54.15% - 63.66% of the total messages. Such a high concentration of messages containing popular documents suggests that using application-level multicast [8, 11, 17, 22] for popular documents would yield significant savings in both bandwidth and server load.

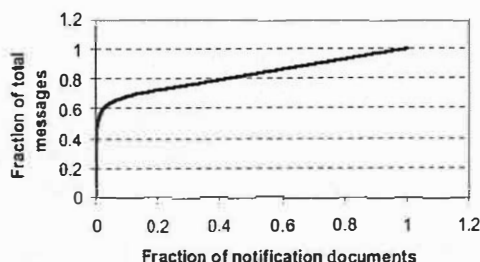


Figure 6: Cumulative distribution of notification messages to documents (for Aug 21, 2000).

A possible optimization is to distribute a set of caches over the Internet to form an overlay multicast tree rooted at the notification server. When a notification message needs to be sent to multiple recipients simultaneously, it can be sent over the overlay tree and also stored at the caches that it traverses. These caches can help in offloading the retransmission work (say, due to a client coming online) from the server: when the same copy of notification needs to be sent at a later time, the caches

closest to the receiver can forward the message

Note that even though the current notification traffic is not significant, as the popularity of notification services increases, bandwidth usage will become an important factor for scaling the notification system. Consequently, optimizations such as application-level multicast will become more important.

We also observed that the concentration of notification messages to documents becomes less pronounced as the number of the documents considered increases. For example, the top 7.6% - 42.0% of the documents account for 80% of the total messages, and the top 45.1% - 71.0% of notifications account for 90% of the total messages. This implies that a large performance benefit can be obtained by multicasting only the most popular notification documents.

### 4.3 User Behavior Analysis

We now study two aspects of user behavior: (i) the spatial locality of user interest, and (ii) the distribution of load that users place on the server.

#### 4.3.1 Spatial Locality

Spatial locality of user interest is about determining whether people in the same geographical region tend to receive similar notification content. To carry out our analysis we take the following approach. We define a notification message to be locally shared if at least two users in the same cluster receive the notification. We compare the degree of sharing using geographical clustering and four random clusterings. In the geographical clustering case, clients in the same city are clustered together. In the random clustering case, clients are clustered randomly with the cluster size being the same as in geographical clustering. We obtained the geographical location of users using a registration database which contains zip code information for each user. The zip code information is not clean — some users supplied invalid zip codes; we filter out all the zip codes that are not 5 digits. 14% of the users supplied such invalid zip codes. In the remaining entries, it is still possible to have zip codes that do not match the actual user location, but the fraction is likely to be small. Furthermore, when computing the degree of local sharing, we exclude the cities to which fewer than 100 notification messages were sent over the course of the week.

As shown in Figure 7, clients residing in the same city have significantly more sharing in notification content compared to the clients picked at random. We also compared geographical clustering with three other random clusterings and observed similar results. The higher degree of sharing in notification messages for clients in the same geographical region indicates that localized services are popular for notification services. For example, people living in New York are interested in receiving notification messages about weather or events in New York. The geographical locality in notification content implies that placing servers (i.e., either notification server replicas or servers in an overlay network that provide application-level multicast) close to popular geographical clusters can be useful in reducing network load.

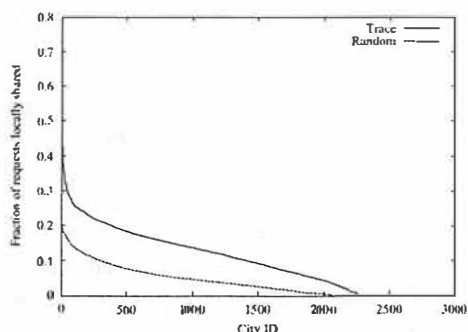


Figure 7: Compare the local sharing between random clients and clients that are geographically close together.

#### 4.3.2 Load distribution of different users

On average, we observed that a user receives 2.3 notification messages containing a total of 0.2 KBytes per day, and 16.1 notification messages containing 1.4 KBytes of data per week. There is a significant variation in the clients' usage — during the week that we studied, some clients received over 1000 messages (containing as high as 0.1 MB of data), while other clients received fewer than 10 messages containing as little as a few hundred bytes of data.

Figures 8 and 9 show the total number of messages and the total number of bytes received by different users on a log-log scale, respectively. Both curves fit very well with a straight line (i.e., follow Zipf-like distribution), except at the tail where there is a sudden drop. We compute the values of  $\alpha$  using least-square fitting, after excluding the sharp drop at the tail. The value of  $\alpha$  is 0.4437 when usage is defined as the number of messages; when usage is defined as the number of bytes, its value is 0.4567.

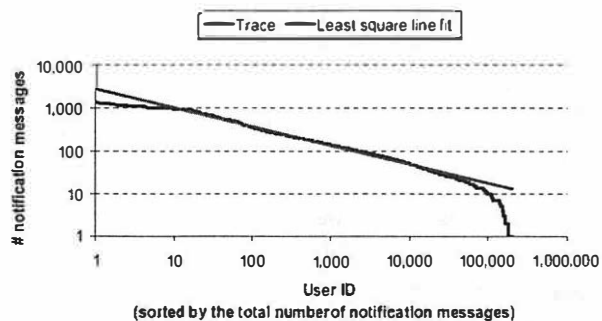


Figure 8: The total number of notification messages received by different users.

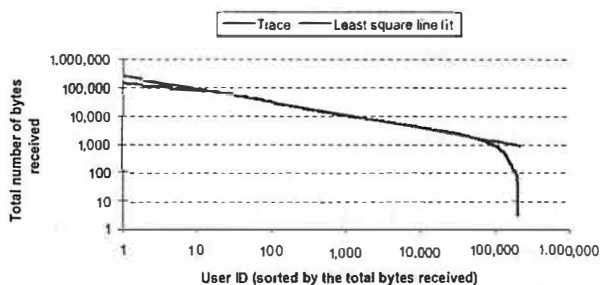


Figure 9: The total number of notification bytes received by different users.

To further study how usage is distributed across different clients, we plot the cumulative distribution of client usage in Figure 10. As the figure shows, the top 5% of the clients received 28% of the notification messages, and 25% of the notification bytes; the top 10% of the clients received 40% of the notification messages, and 38% of the notification bytes. It is clear that a small fraction of users consume a significant fraction of the system and network resources. It is also interesting to note that the CDF curves are similar for the two different ways of defining usage. The similarity of the curves shows that each user receives a similar number of bytes per message.

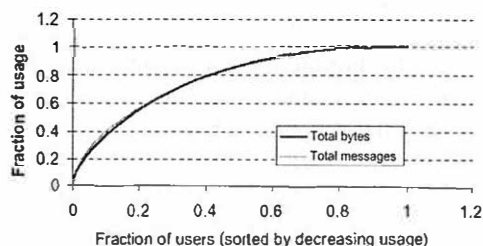


Figure 10: Cumulative distribution of different clients' usage.

The cumulative load imposed by all users (in terms of number of messages and the number of bytes sent by the servers) is shown in Figure 11. The figure shows that the number of messages and the number of bytes are fairly constant during weekdays but exceed the number sent

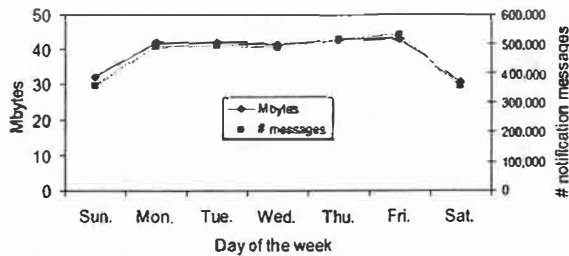


Figure 11: Number of bytes and messages served by the notification servers during the days in the week

during the weekend. This confirms what one would expect, i.e., information alerts are more frequently generated when people are working.

#### 4.4 Summary

Our analysis shows that notification messages are small, popular documents account for a significant fraction of the messages, and there exists a high degree of sharing in geographical regions. System designers need to develop transport protocols that can send such messages in a reliable, efficient and secure manner. For example, an overlay network consisting of geographically placed caches along with application-level multicast can reduce the total network bandwidth requirements and server load. We also observed that there is a significant variation in clients' usage of notification services. Service providers can design pricing plans according to the needs of the clients and also specialize content based on geographical location.

### 5 Browser Log Analysis

In this section, we present our browser logs analysis. In our earlier work, we performed analyses on document content and popularity, distribution of user sessions, and system load [1]. For the sake of completeness, we first summarize the major findings of our previous analysis, and then study the temporal stability and spatial locality of user accesses, as well as the distribution of the load placed by different users on the web server.

#### 5.1 Summary of previous analysis

In [1], we analyzed the browser log collected during the period from August 15, 2000 through August 26, 2000. During this time the web server received 1.6 – 3.2 million requests per day from 64,000 – 98,000 distinct clients. Below is a synopsis of our major findings:

1. The distribution of document popularity does not closely follow Zipf-like distribution, where a document is defined as a unique URL or as a unique URL and parameter pair. The majority of requests are concentrated on a small number of documents. In particular, we found that 0.1% – 0.5% of the documents (i.e., approximately 121 – 442) account for 90% of the requests.
2. More than 60% of the pages accessed at the web server are due to offline PDA users and less than 7% of the accesses are due to wireless clients; the remaining accesses are due to desktop clients for registration and customization services.
3. Our analysis for the distribution of reply sizes showed that most of the replies to wireless clients are less than 3 KBytes. For offline clients, most of the replies are less than 6 KBytes. The reply size distribution for the two types of clients is similar.
4. Our user session analysis showed that users tend to have short sessions when interacting with the web site: 95% of the sessions were less than 3 minutes. We empirically determined the session-activity threshold to be somewhere between 30 to 45 seconds (i.e., if no request is received from a client for such a duration, it implies that the old session has ended).
5. Our category analysis showed that stock quotes, news, and yellow pages are the top categories accessed by wireless clients. For offline clients, help is the most popular category followed by news and stock quotes.
6. We observed that the relative importance of different categories did not change between weekdays and weekends (except stock quotes and sports). However, the amount of data accessed over the weekend drops by approximately 45%.

These findings have the following performance implications:

1. The high concentration of requests to popular documents in the browser log implies that caching the results of popular queries would be very effective in reducing the web server load.
2. Since most replies sent to wireless and offline users are small (3 – 6 KB), the wireless web server should be highly optimized in sending short replies, e.g., optimizing TCP slow start and re-start [15, 23] can be useful in this environment.

3. Our heuristic, based on user session analysis, to determine the session-inactivity period can be useful to wireless service providers who want to reclaim IP addresses. Our analysis showed that IP addresses may be reclaimed more quickly than the time period determined in earlier work [12].

## 5.2 New Analysis

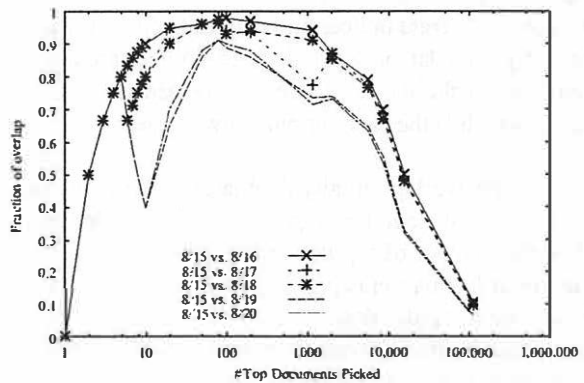
We now present a user-based analysis of the browser logs (based on the unique identifier associated with each browse request). We examine temporal stability, spatial locality, and usage variation across different users.

### 5.2.1 Temporal Stability

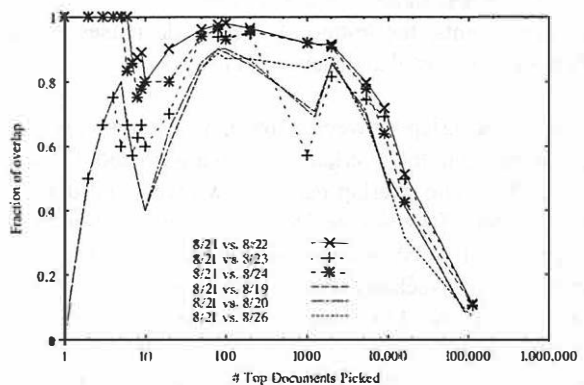
In the section, we analyze whether users are interested in a similar set of documents on different days. To answer this question, we pick the  $N$  most popular documents from each day, and compare the extent of the overlap. Since all the web pages are dynamically generated, a document is defined as a combination of a unique URL name and the query parameters (i.e., two requests with the same URL with different parameters are considered as different document requests). We will use the terms document and query interchangeably in this section.

First we study the requests from all users, i.e., including wireless, offline, and desktop users. Figure 12 (a) and (b) plot the overlap between weekdays August 15 (Tuesday) and August 21 (Monday) versus other days (i.e., both weekend days and weekdays) (In Figure 12 (a) and (b), the curves with points are for pairs of weekdays, and those without points are for a weekday and weekend.) Figure 13 plots the overlap between weekend days. Note that the x-axis data value for the top  $N$  case does not always correspond to exactly  $N$  in the graphs. The reason is that when we consider the top (say) 100 documents, the next few documents after these documents may also have the same frequency as the 100<sup>th</sup> document; since we include these documents as well for the “top 100” data point, it sometimes results in a small mis-match of the plotted points.

Looking at Figure 12 (a) and (b), we make the following observations: first, the overlap between different days is significant. For example, the overlaps are over 80% for the top 100 documents, and mostly over 70% for the top 1000 documents. This indicates that the set of popular queries remains relatively stable, and suggests that we can cache a stable set of popular query results or opti-



(a) Overlap between a weekday versus other days.



(b) Overlap between another weekday versus other days.

Figure 12: Temporal stability of document ranking for weekdays

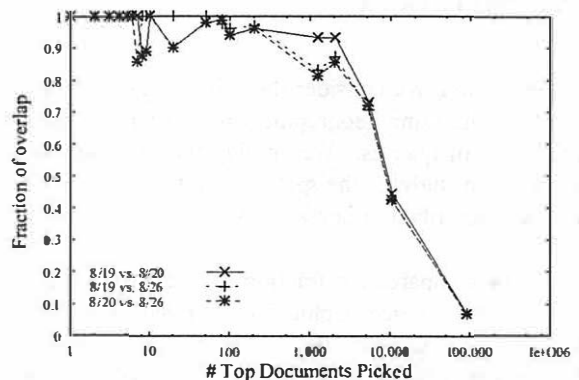


Figure 13: Temporal stability of document ranking between weekend days

mize the data layout to improve the performance of these queries. For example, workload-based techniques can be used to generate indices and materialized views automatically for a database [2]; these techniques are largely applicable if the database query workload is relatively stable (which is the case for our browser queries).

Second, the overlap initially fluctuates with the increasing number of documents picked, and then decreases when the number of top documents picked is over 100. The initial fluctuation is probably due to the fact that although very popular documents tend to remain popular, their relative ranking does change over time. However, as we further increase the number of documents, we may include some less popular documents. Since these documents are less likely to remain popular than very popular documents, the temporal overlap decreases. This phenomenon was also observed in [16].

Third, the overlap between pairs of weekdays is generally higher than the overlap between a weekend day and a weekday. The overlap between two weekend days is even higher. This is consistent with our intuition, and suggests that we should use past weekday workload to predict future weekday workload, and likewise use past weekend workload to predict future weekend workload.

We also examine the requests coming from only the wireless users, and find the results are very similar. As before, the set of popular queries remains stable over time. The stability is especially high when we consider the most popular queries. In addition, there is a significant difference between the access pattern on weekdays versus that on weekends.

### 5.2.2 Spatial locality

In this section, we consider the following question: do people in the same geographical region tend to issue a similar set of queries. We employ the same approach as is used in studying the spatial locality for notification services (described in Section 4.3.1).

Figure 14 compares the fraction of documents that are shared within a geographical cluster and within four random clusters, when we consider requests from all the users (excluding users with invalid IDs). The figure shows that the curve for the geographical clusters overlaps with those for random clusters. This overlap indicates that the degree of sharing between geographical clustering and random clustering is comparable, and the correlation between users' interest in brows-

ing over wireless channels and their geographical location is weak.

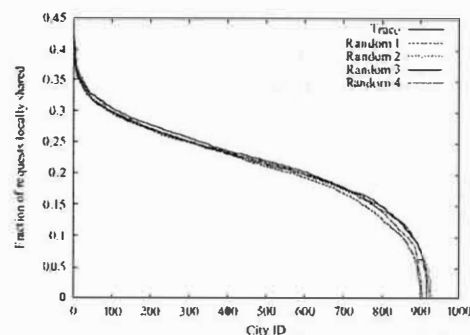


Figure 14: Local sharing between random sets of clients and clients that are geographically close together.

A possible explanation for the weak correlation is that the popular browse content has global interest. In particular, as mentioned in Section 5.1, 0.1% - 0.5% of the URL and parameter combinations (i.e., about 121 - 442 unique combinations) account for 90% of the requests. With such a high concentration of user interest on a few documents, even when clients are picked at random, they share many requests; therefore, the geographical locality becomes insignificant. A similar phenomenon has been observed in a study of a popular news server [16], where the authors observed that the significance of domain membership becomes diminished during a popular event. A major distinction between that study and ours is the way in which users are clustered: in that study, users are clustered based on their DNS names, whereas in our study we cluster users based on their geographical region, e.g. the city in which they reside.

A natural question follows - why is there such a high concentration of interest in popular documents that even when clients are picked at random they share many documents? Examination of the most popular URLs and parameters shows that they include the front pages for email login, news, sports, weather, lottery, and the signup application, as well as some popular stock quote queries. Intuitively, these queries are very popular to all users regardless of their physical locations.

The lack of geographical locality implies that the web server's content can be replicated without keeping in mind the geographical location of the clients.

We performed the same spatial locality analysis to requests issued only by wireless clients. Figure 15 summarizes the results. With geographical clustering, wireless clients have slightly more sharing of documents than with random clustering; however, the distinction between the two clusterings is much less significant than



the difference observed for notification documents. This result suggests that using geographical locality of wireless users as input for optimizing performance (or providing content) will yield limited success.

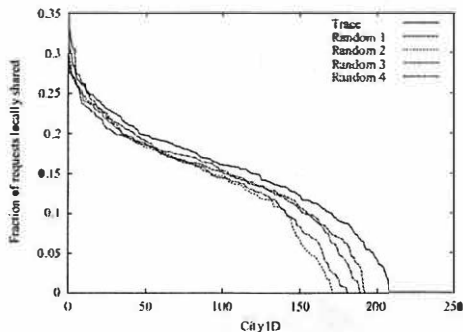


Figure 15: Comparison of local sharing between random sets of wireless clients and wireless clients that are geographically close together.

### 5.2.3 Load distribution of different users

In this section, we study the distribution of loads placed on the web server by different users. Our earlier analysis [1] examined the difference in load distribution between wireless users and offline users. Now we look at the load distribution at a more fine-grained level — at a per-user level.

Figure 16 and Figure 17 show the total number of accesses and total number of data requested by different clients, respectively (users with invalid identifiers were discarded). As the figures show, there is a significant variation in the load placed by different users on the web server: some users request several orders of magnitude more documents/data than other users. The accesses from only the wireless clients reveal similar property. Thus, service providers can consider designing different pricing plans that to cater to the widely varying needs of different users.

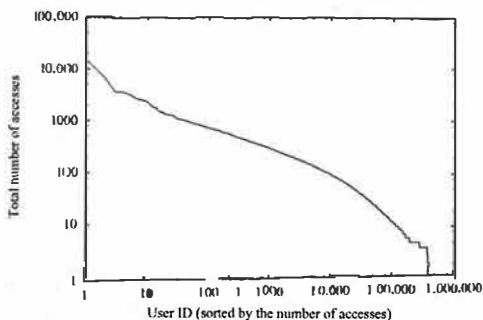


Figure 16: Total number of accesses made by different users.

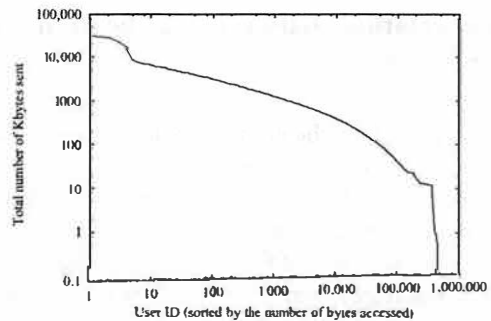


Figure 17: Total number of data received by different users.

Figure 18 shows the inter-arrival time between requests coming from the same user. The requests generated from the offline users are much more bursty than those from the wireless users: 97% of the requests from the offline users have 1 second or less inter-arrival time, whereas only 9% of the requests from the wireless users have comparable inter-arrival time. We observe very bursty traffic for offline PDA users because their requests are generated by the downloader program rather than a human being; these users also generate significantly more requests than wireless users. If not handled appropriately, such bursts can delay wireless users unnecessarily. The web site designers can address this problem in a number of ways. For example, they can provide higher priority to wireless users or restrict the burst of offline user requests to a few front-door servers (servers that handle incoming HTTP requests). An orthogonal efficiency issue that needs to be addressed is the synchronization protocol for PDAs, i.e., instead of sending a large number of small requests, the synchronization protocol could batch all these requests into a single request and reduce the server load and roundtrip latency.

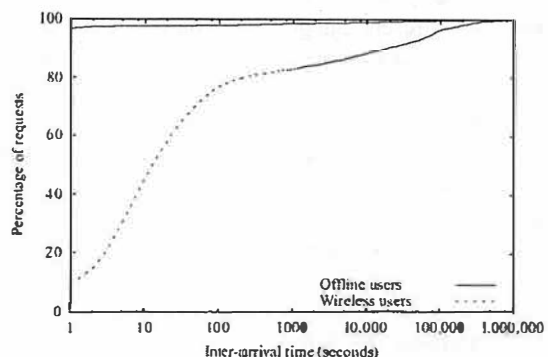


Figure 18: CDF of inter-arrival time between consecutive requests from the same user.

## 6 Correlation between notifications and browsing

Having studied both the notification logs and the browse logs, it is useful to understand whether there is any correlation between the browsing and notification activities of users. We are interested in answering questions such as: (i) do users utilize one of the services significantly more than other services, and (ii) does their interest in particular content categories differ across the two services. We use the notification and browser logs, both spanning from August 15, 2000 through August 26, 2000 for the following analysis.

### 6.1 Correlation in the amount of usage

Figure 19 shows the average number of notification messages versus the number of browse requests, and the average number of browse requests versus the number of notification messages. There is little correlation between the two variables: the number of notification messages fluctuates widely with the number of browse requests; similarly, the number of browse requests also shows no obvious trend with respect to the number of notification messages. The correlation coefficient between these two variables is 0.265 when considering all users, and 0.125 when considering only wireless users. The low correlation coefficients implies that web site designers cannot predict a user's browsing activity based on his/her notification activity, and vice versa.

### 6.2 Correlation in popular content categories

We now look at the question whether users are interested in a similar set of content categories across the two services. To answer this we take the following approach: first, we classify notification messages and browsing accesses into different categories. (The details of categorizing notifications are described in Section 4.2, and the details of categorizing browse accesses are described in our earlier work [1].) Then for each individual user, we pick the top  $N$  content categories in browsing and top  $N$  content categories in notification (if the next few categories after the  $N^{th}$  category have the same frequency of access as the  $N^{th}$  category, we include those categories as well for the top  $N$  case).

Figure 20 shows the percentage of users who have at least some overlap between their top  $N$  browse and notification categories. The degree of overlap is much higher when we consider wireless users only. For example, for

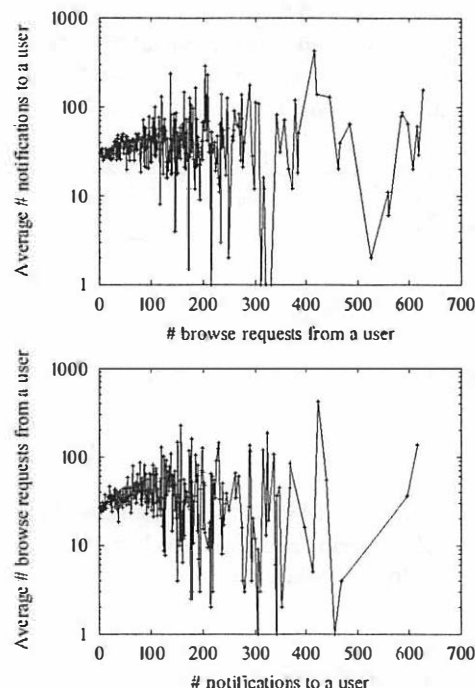


Figure 19: Correlation between the number of browse requests and notifications of wireless users.

the top 3 categories, the percentage of overlapped users is less than 10% when considering all the users, and around 50% when considering only the wireless users. On the other hand, even when considering wireless users only, the number of overlapped users is never more than 65%.

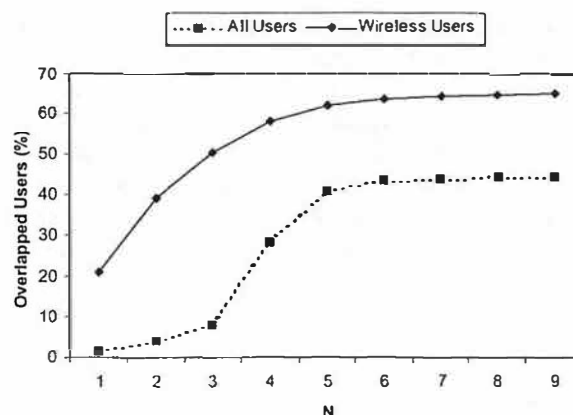


Figure 20: Number of users who have overlap between their top  $N$  browsing categories and top  $N$  notification categories.

We now compare the extent of the overlap by varying  $N$  from 1 to the total number of categories. The results are shown in Figure 21. The figure shows the average percentage of overlap between two categories, where the average overlap is computed as follows:

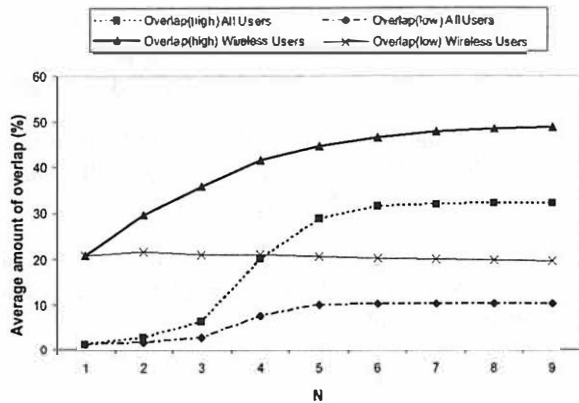


Figure 21: Correlation between the number of browse requests and notifications of wireless users.

$$overlap_{high} = \frac{\sum_i \frac{\#categories\ overlapped\ for\ user_i}{\min(N, \min(BC, NC))}}{\text{relevant users}}$$

$$overlap_{low} = \frac{\sum_i \frac{\#categories\ overlapped\ for\ user_i}{\min(N, \max(BC, NC))}}{\text{relevant users}}$$

where  $BC$  denotes the number of browse categories,  $NC$  denotes the number of notification categories, and relevant users refers to those users that have at least one browse record and one notification record in the respective logs. We show the results for only the top 9 categories, since the values beyond that are stable.

Essentially these ratios compute the percentage of overlap for each individual user, and then take the average of these percentages over all wireless users or all users. Since not all users have at least  $N$  browsing or notification categories, we compute  $overlap_{high}$  and  $overlap_{low}$ , where the former computes the percentage of overlap by using the minimum of  $BC$  and  $NC$ , and the latter uses the maximum of  $BC$  and  $NC$ . The figure shows that the amount of overlap is considerably higher when considering only wireless users. For example, for the top three categories, the overlap is less than 7% when considering all users. In comparison, for wireless users, the  $overlap_{low}$  and  $overlap_{high}$  values are 21% and 36%, respectively. We also observe that the effect of increasing  $N$  is small. Even when  $N$  is 8, the percentage of overlap is less than 50% for wireless users.

The above results indicate that wireless users have moderate correlation in the way they use browse and notification services. In comparison, the correlation is much lower when considering all users. This is because the most popular browsing categories for desktop users are

sign-up services, direction, and general help, whereas notification is usually not used to deliver these types of content. On the other hand, some wireless users are interested in both browsing and receiving notifications about emails, stock quotes, personalization, news and sports. However, the degree of correlation is limited, and service providers cannot solely rely on a user's notification profile to determine what content he/she may be interested in browsing.

## 7 Conclusions

Internet access via small handheld devices is expected to increase tremendously in the next few years. In this paper, we analyzed the access patterns of a large web site designed primarily for wireless and handheld mobile devices. The web site provides both browse and notification services. To our knowledge, this is a first-of-a-kind study that analyzes notification services. It is also first in analyzing user behavior using a commercial web site. We believe this is an important first step in the direction of understanding the dynamics of wireless Internet services.

## 8 Acknowledgements

We thank Mike Spreitzer for carefully reading our paper several times and helping us improve its quality significantly. We also thank the anonymous reviewers for their detailed comments and suggestions that improved the paper in several aspects.

## References

- [1] A. Adya, P. Bahl, and L. Qiu. Understanding the Browse Patterns of Mobile Clients using a Popular Web Server. *ACM SIGCOMM Internet Measurement Workshop*, November 2001.
- [2] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated Selection of Materialized Views and Indexes for SQL Databases. In *Proceedings of the 26th International Conference on Very Large Databases (VLDB00)*, Sep 2000.
- [3] M. Arlitt and T. Jin. Workload characterization of the 1998 World Cup web site. *IEEE Network*, pages 30 – 37, May/June 2000.
- [4] M. F. Arlitt and C. L. Williamson. Internet Web Servers: Workload Characterization and Performance

- Implications. In *IEEE/ACM Transactions on Networking*, pages 631–645, 1997.
- [5] A. Balachandran, G. Voelker, P. Bahl, and V. Rangan. Characterizing User Behavior and Network Performance in a Public Wireless LAN. In *Proceedings of ACM SIGMETRICS '02*, June 2002.
  - [6] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. Changes in Web Client Access Patterns. *World Wide Web Journal*, 1999.
  - [7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of INFO-COMM '99*, March 1999.
  - [8] Y. Chu, S. Rao, S. Seshan, and H. Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of ACM SIGCOMM 2001*, August 2001.
  - [9] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW Client-based Traces. *Technical Report TR-95-010*, April 1995.
  - [10] S. Glassman. A Caching Relay for the World Wide Web. In *Proceedings of 1st WWW Conference*, May 1994.
  - [11] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and Jr. J. W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network, 2000.
  - [12] T. Kunz, T. Barry, X. Zhou, J.P. Black, and H.M. Mahoney. WAP Traffic: Description and Comparison to WWW Traffic. *ACM Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, August 2000.
  - [13] J. C. Mogul. The Case for Persistent-Connection HTTP. In *Proceedings of ACM SIGCOMM 95*, August 1995.
  - [14] N. Nishikawa, T. Hosokawa, Y. Mori, K. Yoshida, and H. Tsujia. Memory-based Architecture for Distributed WWW Caching Proxy. In *Proceedings of 7th WWW Conference*, April 1998.
  - [15] V. N. Padmanabhan and R. Katz. TCP Fast Start: A Technique for Speeding up Web Transfers. In *Proceedings of IEEE Globecom '98*, November 1998.
  - [16] V. N. Padmanabhan and L. Qiu. The Content and Access Dynamics of a Busy Web Site: Findings and Implications. In *Proceedings ACM SIGCOMM 2000*, August 2000.
  - [17] D. Pendarakis, S. Shi, D. Venna, and M. Waldvogel. ALMI: An Application Level Multicast Infrastructure. In *Proceedings of USITS 2001*, March 2001.
  - [18] D. Tang and M. Baker. Analysis of a Metropolitan-Area Wireless Network. In *Proceedings of ACM MobiCom 99*, pages 13–23, August 1999.
  - [19] D. Tang and M. Baker. Analysis of a Local-Area Wireless Network. In *Proceedings of ACM MobiCom 2000*, August 2000.
  - [20] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, M. Brown T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organizational-based Analysis of Web-Object Sharing and Caching. In *Proceedings of USITS '99*, October 1999.
  - [21] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, M. Brown T. Landray, D. Pinnel, A. Karlin, and H. Levy. On the Scale and Performance of Cooperative Web Proxy Caching. In *Proceedings of SOSIP '99*, Dec. 1999.
  - [22] H. Yu, L. Breslau, and S. Shenker. A Scalable Web Cache Consistency Architecture. In *Proceedings of ACM SIGCOMM '99*, 1999.
  - [23] Y. Zhang, L. Qiu, and S. Keshav. Speeding up Short Data Transfers: Theory, Architectural Support, and Simulation Results. In *Proceedings of NOSS-DAV'2000*, June 2000.







# THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

## SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

## Member Benefits

- Free subscription to *login*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *login* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

## Supporting Members of the USENIX Association

Interhack Corporation	Smart Storage, Inc.
Lucent Technologies	Sun Microsystems, Inc.
Microsoft Research	Sybase, Inc.
Motorola Australia Software Centre	Taos: The Sys Admin Company
The SANS Institute	TechTarget.com
Sendmail, Inc.	UUNET Technologies, Inc.

## Supporting Members of SAGE

Certainty Solutions	New Riders Publishing
Collective Technologies	O'Reilly & Associates Inc.
ESM Services, Inc.	Ripe NCC
Lessing & Partner	Taos: The Sys Admin Company
Microsoft Research	Unix Guru Universe
Motorola Australia Software Centre	

For more information about membership, conferences, or publications,  
see <http://www.usenix.org/>

or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA  
Phone: 510-528-8649 Fax: 510-548-5738 Email: [office@usenix.org](mailto:office@usenix.org)

ISBN 1-880446-00-6